MASTER IN COMPUTER SCIENCE

A validation tool for the UEML approach

Mahiat, Jérémy

Award date:
2006

A Validation tool for the UEML Approach

Jérémy Mahiat

Mémoire présenté en vue de l'obtention du grade de
Maître en Informatique

Année académique 2005 - 2006

Abstract

The current economical environment constrains enterprises to be more flexible and reactive in order to be able to anticipate and adapt to frequent changes they have to face. In order to master these changes, enterprises have to determine, as clearly as possible, their way of working and their environment. A fundamental tool for that purpose is Enterprise Modelling (EM). EM has an extremely large scope that gave rise to many different Enterprise Modelling Languages (EMLs). As enterprises deal with many different incompatible but interrelated models and as they work more and more together, a real need for EMLs interoperation appeared.

UEML (Unified Enterprise Modelling Language) sets itself up as a solution to this need. The project aims to create a federator language. It developed a method to analyze EMLs and to capture the knowledge from these analyses in an ontology. Tools are provided in order to facilitate the process. The main goal of our work is to describe the solutions brought by UEML and to provide a new tool that helps to validate the content of the analyses. In order to build this new tool called "UEML Validator", we formalized the method by establishing 65 constraints that restrict how the ontology can be populated and ensure it to be more consistent. By interpreting the "UEML Validator" results, we make recommendations for another UEML tool in construction.

Keywords: Unified Enterprise Modelling language, UEML, Enterprise Modelling Language, interoperation, validation, tool, ontology, OWL, Prolog, meta modelling, syntax and semantics of modelling languages.

Résumé

L'environnement économique actuel oblige les entreprises à être plus flexibles et plus réactives afin de pouvoir anticiper et s'adapter aux changements fréquents auxquels elles doivent faire face. Pour maîtriser ces changements, les entreprises doivent connaître au mieux leur manière de travailler ainsi que leur environnement. Pour ce faire, un outil fondamental est la Modélisation d'Entreprise (ME). La ME a un champ d'application extrêmement large, ce qui a donné naissance à de multiples Langages de Modélisation d'Entreprise (LME). Les entreprises travaillent avec beaucoup de modèles inter-dépendants mais incompatibles et collaborent de plus en plus souvent, ce qui a généré un besoin d'interoperabilité des LMEs.

UEML (Unified Enterprise Modelling Language) est une solution à ce problème. Le projet a pour but de créer un langage fédérateur. Il a développé une méthode pour analyser les LMEs et pour garder les connaissances acquises lors de ces analyses dans une ontologie. Des outils ont été développés pour faciliter le processus. Le but principal de notre travail est de décrire les solutions apportées par UEML et de fournir un outil qui permet de valider le contenu des analyses. Pour réaliser cet outil ("UEML Validator"), nous avons formalisé la méthode en établissant 65 contraintes qui limitent la manière dont l'ontologie peut être peuplée et qui en assurent la cohérence. L'interprétation des résultats d'"UEML Validator" nous a permis de faire des recommandations pour la construction d'un autre outil pour UEML.

Mots-clefs: Unified Enterprise Modelling language, UEML, Langage de modélisation d'entreprise, interopération, validation, outil, ontologie, OWL, Prolog, meta modelisation, syntaxe et sémantique des languages de modélisation.

Preface

This report is mainly the result of a nine semester project carried out in Fall 2005 in the *Universidad Politecnica de Valencia* (Spain) and continued in the University of Namur (Belgium) in spring 2006. In Spain, I worked under the supervision of Pr. Andreas Lothe Opdahl who is one of the main actor of the UEML 2.0 project. UEML 2.0 is an ongoing, exploratory and fundamental project that is part of InterOp, a Network of Excellence supported by the European Commission. Names and methods changed and evolved during the whole year.

I would like to express my gratitude to Professor Andreas Opdahl for his guidance and availability and for the general help he provided me during my internship.

A special thanks goes to my supervisor, Professor Patrick Heymans, for his continuous help and his numerous advices during the writing. I also want to thank him for the opportunity he offered me to work in another University on a really interesting subject.

I would also like to thank Doctor Raimundas Matulevičius who patiently read the different version of my work and made me improve it.

Je voudrais également remercier Baudouin pour son aide durant la réalisation de ce mémoire.

Je voudrais finalement remercier ma famille qui m'a toujours encouragé et m'a permis de réaliser mes études et ce stage à l'étranger. Merci aussi à Sophie pour tout ce qu'elle m'a apporté.

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

BWW	Bunge-Wand-Weber representation model of information systems
EEML	Extended Enterprise Modelling Language
EM	Enterprise Modelling
EML	Enterprise Modelling Language
GRAI	Graphs with Results and Actions Inter-related
GRL	Goal-Oriented Requirement Language
IEM	Integrated Enterprise Modelling
IS	Information Systems
JPL	Java Prolog Library
ML	Modelling Language
OCL	Object Constraint Language
OWL	Web Ontology Language
RDF	Resource Description Framework
UEML	Unified Enterprise Modelling Language
UML	Unified Modelling Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

The current economical environment constrains enterprises to be more flexible and reactive in order to be able to anticipate and to adapt the frequent changes they have to face. These changes of technological, sociological or economical nature can have deep impacts on the enterprise structure. Mastering these changes can be crucial success factor for many enterprises working in a competitive area.

In order to master these changes, enterprises have to determine as clearly as possible their way of working and the environment in which they are working. One of the fundamental solutions for that purpose is Enterprise Modelling (EM). EM enables companies to externalize information concerning the many facets of the enterprise, for instance in order to describe their organisation or their operational processes. The goals are very broad; it can be used to analyse and to restructure enterprises for better results, to compare different possible scenarios to find the best solution or to teach new people cooperating with the company.

EM can be used in every sector, producing goods or providing services. It can also be used to model many different enterprise areas such as organisation, resources, process, information, requirements, goals or strategy. This extremely large scope gave rise to many different Enterprise Modelling Languages (EML).

The different enterprise models inside one enterprise are interrelated. Enterprises cooperate more and more and need to exchange their knowledge captured in different models. But most EMLs are implemented in tools with proprietary terminology and modelling constructs. The different enterprise models are thus incompatible. Therefore a real need for EMLs interoperation has appeared.

Unified Enterprise Modelling Language sets itself up as a solution to this need. Two projects (UEML 1.0 [UEMa] and UEML 2.0 [UEMb]) aim to create a federator language that enables to integrate existing modelling languages and to support their comparison, consistency checking, update reflection, view synchronization and, eventually, model-to-model translation across modelling language boundaries. These two projects are related but differ in their goals and approaches. The first one was made with three different languages while the second one aims to be able to easily incorporate new languages. The second version tries to analyse the different EMLs by building an ontology about what they represent in the real world.

We contribute to the UEML 2.0 project which is still in progress. It developed a method to analyze EMLs and to capture the knowledge from these analyses. Tools are provided in order to facilitate the process and to enable one to use this knowledge.

The main goal of our work is to describe the solutions brought by UEML and to provide a new tool that helps to validate the content of EMLs analyses. In order to build this new tool called "UEML Validator", we formalise the analysis method by establishing 65 constraints restricting how the ontology can be populated. These constraints are general and do not take the context into account.

1.2 About the document

The document is divided into three parts. First, the background explains what EMLs are and presents GRL, a particular EML used as an example throughout this work. This part shows where the need for interoperation comes from and different alternatives to tackle the problem. We present the two UEML versions before going more deeply in details about the notions on which UEML 2.0 relies. Its template and meta-meta-model will be explained.

In the contribution part, we depict the different tools that support the UEML 2.0 approach. The requirements for each of them will be stated. A detailed presentation of the use and implementation of the tool we created for the validation of analyses will be presented in a chapter.

The third part will provide an application and an evaluation of the concepts and tools introduced in the two first parts. An analysis of GRL using the UEML 2.0 approach and the different tools already available will be presented.

Finally, general conclusions about this project and suggestions for future work are given.

These three parts are composed of the following chapters:

Chapter 2 introduces the notion of EML, explains the need for interoperation and explains the use of a particular EML: GRL.

Chapter 3 provides an overview and a comparison of the two versions of UEML.

Chapter 4 explains how the UEML 2.0 template works. The template is the technique used by UEML 2.0 to analyse languages.

Chapter 5 deals with ontology in general, with the OWL language that enables to describe them and with Protégé that gives the opportunity to manage OWL ontologies.

Chapter 6 depicts the UEML 2.0 Meta-meta-model. This model has been written in OWL and relies on the BWW Model. The meta-meta-model is the ontology used to capture what languages can represent.

Chapter 7 is the first chapter of the contribution part. It shows the utility of tools in the UEML 2.0 approach and give an overview of them.

Chapter 8 focus on the program we made: "UEML Validator". It explains its purpose, how it works and the constraints it checks.

Chapter 9 is the first chapter of the application part. It provides the results obtained from the use of UEML Validator. It also gives interpretation of these results.

Chapter 10 gives a typical scenario of applying UEML 2.0 techniques and tools on the GOAL GRL construct.

Chapter 11 provides a conclusion of the work by giving its limitations and future developments to be reached.

Part I

Background

Chapter 2

Enterprise Modelling Languages

The objective of this chapter is to explain what an Enterprise Modelling Language (EML) is and what are its main characteristics. We show why it is important to make them interoperate and will also introduce a specific EML called GRL (Goal-oriented Requirements Language).

2.1 Enterprise modelling

In order to know what an enterprise modelling language (EML) is, we first have to know what enterprise modelling (EM) is. [Ver96] defines it as follow:

> "Enterprise modelling is the set of activities or process used to develop the various parts of an enterprise model to address some desired modelling finality. It can also be defined as the art of "externalising" enterprise knowledge, i.e. representing the enterprise in terms of its organisation and operations (e.g. processes, behaviour, activities, information, object and material flows, resources and organisation units, and system infrastructure and architectures)."

The use of EM is represented on Figure 2.1. In [PD02], it is explained as a finality that makes explicit facts and knowledge that add value to the enterprise or can be shared by business applications and users for the sake of improving the performance of the enterprise. The primary goal of EM is not only to be applied for better enterprise integration but also to support analysis of an enterprise, and more specifically, to represent and understand how the enterprise works, to capitalize acquired knowledge and know-how for later reuse, to design (or redesign) a part of the enterprise, to analyse some aspects of the enterprise (e.g., economic analysis, organisation analysis, qualitative or quantitative analysis, requirements analysis, ...), to simulate the behaviour of (some part of) the enterprise, to make better decisions about enterprise operations and organisation, or to control, coordinate and monitor some parts of the enterprise.

2.2 Enterprise modelling languages

EMLs are thus (usually visual) languages that allow one to do EM as just described. The Generalised Enterprise Reference Architecture and Methodology (GERAM) [oAfEI99] defines them in those words:

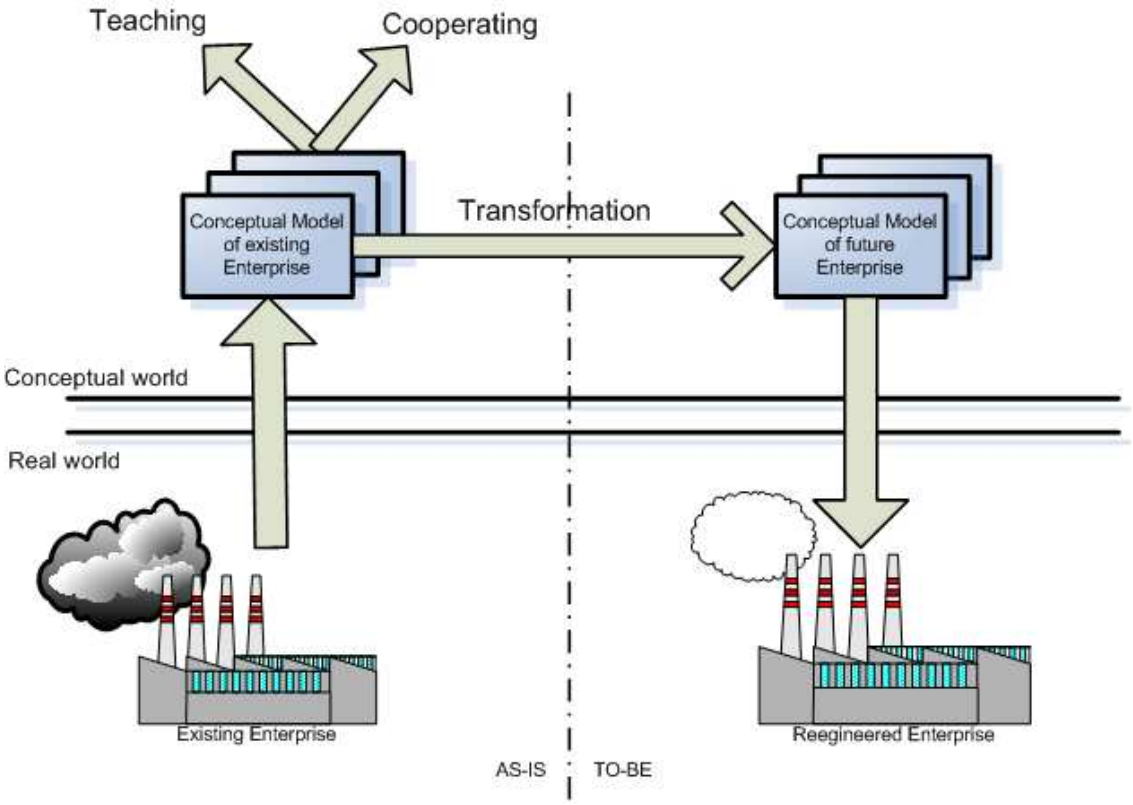


Figure 2.1: Some Enterprise Modelling uses.

> "EMLs define the generic modelling constructs for EM adapted to the needs of people creating and using enterprise models. In particular EMLs will provide construct to describe and model human roles, operational processes and their functional contents as well as the supporting information, office and production technologies."

Some languages are fully qualified as EMLs because they are historically related to the field of EM; but many other languages are used for modelling enterprise phenomena even if they were not created specifically for this purpose (e.g., Petri Nets or GRL). In the UEML and in this work in particular, we take both kinds of languages into account.

2.2.1 EML Diversity

EMLs are characterized by one main point: diversity. The story of EMLs related in [PD02] shows the diversity due to the different existing tools: "Within the initiative on Computer Integrated Manufacturing (CIM), Enterprise Modelling was born in the United States at the beginning of the 80s and emerged through large CIM projects, e.g. ICAM (Integrated Computer Aided Manufacturing) led by the US Air Force or CAM-I (Computer Aided Manufacturing - International) via the project Factory Management. In the mid-80s, Europe launched several projects on Enterprise Modelling giving birth to several EMLs (including notably GRAI and CIMOSA). As a result, in the 90s many commercial tools dealing with EM or business process modelling appeared on the marketplace, e.g. ARIS ToolSet, FirstSTEP, METIS, Enterprise Modeller, KBSI tools, CimTool, MOOGO, IMAGIM and many others as well as a myriad of workflow systems, each one with its own modelling environment (Action Workflow, COSA, FlowMark, Lotus Notes, Teamware Flow, Ensemble, WorkParty, ...). "

Models in different domains combinaison

The diversity is due to different factors, but the major one is certainly the fact that EM does not create one monolithic model but an assemblage of models as organisational model, resource model, economic model, functional model, ... Moreover many enterprise languages come from other disciplines, such as software engineering (e.g., UML, Petri Nets), knowledge engineering (e.g., OWL, PSL) and information systems (IS) requirements engineering (e.g., i^*).

Diversity is also due to the fact that EM is used for industrial enterprises (producing goods in a continuous way or not) as well as for enterprises providing services.

Consequences

As a consequence, EMLs are often very different in their nature and this has led to a "Tower of Babel" situation in which the many tools, while offering powerful but different functionalities and semantics, are unable to interoperate and can hardly or not at all communicate and exchange models. It is a serious drawback for awareness, acceptance and wide use of the EM technology in industry which cannot capitalise from previous modelling efforts. This situation hinders true enterprise integration, interoperability, and sharable enterprise knowledge.

[PD02] also shows to which situation this diversity lead:
"A very general first conclusion of the comparison [of EMLs] is the confirmation of a very large number of distinct but overlapping languages. Proposed approaches such as IEM, GRAI, ARIS, CIMOSA, are based on modelling languages to build models. Therefore, the specific objectives (or tasks) that these approaches allow to meet (or perform) are only fully supported if the set of the languages on which they are based provides sufficient expressive power to represent the enterprise aspects needed for specific analysis. Any enterprise methodology has produced a language oriented to the kind of problems to be solved. This state of the art seems to reveal a tendency for approaches combining, in a more or less integrated way, several sub-languages or views (see e.g. CIMOSA,GRAI, ARIS)."

2.2.2 Need to interoperate

The GERAM ([oAfEI99]) showed that EM is a real need for today's enterprises and that it is not possible to choose only one EM tool but one has to use several: "One of the most important characteristics of today's enterprises is that they are facing a rapidly changing environment and can no longer make predictable long term provisions. To adapt to this change enterprises themselves need to evolve and be reactive so that change and adaptation should be a natural dynamic state rather than something occasionally forced onto the enterprise. This necessitates the integration of the enterprise operation and the development of a discipline that organizes all knowledge that is needed to identify the need for change in enterprises and to carry out that change expediently and professionally. This discipline is called Enterprise Engineering.

Previous research has produced reference architectures which were meant to be organizing all enterprise integration knowledge and serve as a guide in enterprise integration programs. The IFIP/IFAC Task Force analysed these architectures and concluded that even if there were some overlaps, none of the existing reference architectures subsumed the others; each of them had something unique to offer. The recognition of the need to define a generalized architecture is the outcome of the work of the Task Force."

Inside one enterprise

For each domain (organisation, resources, process, information, requirements, goals or strategy) that enterprises want to model, a separate EML will be used. Those models are interrelated but

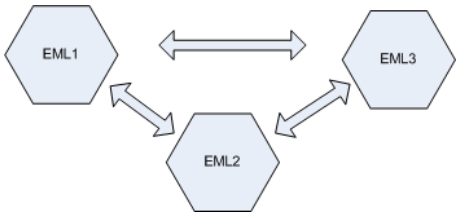


Figure 2.2: Bidirectional translation (adapted from [BPP04]).

not linked. Therefore, there is a real need to elaborate a link between the different models inside an enterprise. Models built in the distinct languages have to be related in some ways and the languages have to be integrated.

Enhancing enterprise cooperation

A second necessity comes from the fact that enterprises work more and more together: it results in knowledge sharing. Indeed, "Enterprise Integration is not anymore only a problem of interconnecting physical and software applications but also requires a global business integration, aiming at the use of the existing or new enterprise resources in order to better achieve the overall business objectives. Things to be integrated and coordinated need to be modelled. Thus, EM is clearly a prerequisite for enterprise integration." [PD02]. Interoperation leads to a problem because most of those EMLs are usually not able to communicate. Indeed most EMLs are implemented in tools with proprietary terminology, modelling construct meaning and format, and template. Each modelling language uses a particular syntax and limited, tool-embedded semantics and graphical notations. Modelling tools manage the same things but do not "talk" to one another.

Solutions

In order to tackle this problem several solutions appear: to invent a new "complete" language, to find translators between each ML couple or to use an intermediate language. The first solution would consist in building a new language covering the domains of a set of the most used languages. Enterprises would just have to translate their models into this new language and everybody would be able to cooperate. The problem is, of course, the translation but also the fact that new needs are likely to appear in the future. Before reaching a consensus about those needs, enterprises will use other languages and the situation might come back at the same point. Moreover, the creation of a language enabling one to make models combining every aspects of the enterprise is an unattainable myth. A software supporting this hypothetical language would be too complex to build and to use.

Another solution is to find a way to enable each couple of EMLs to exchange theirs models. A kind of translator that can transform a model written in one EML into the second EML. Those translators are represented by the bidirectional arrows on Figure 2.2 that shows a combination of EMLs covering only a part of enterprises aspects. This solution is feasible to make EMLs inside an enterprise interoperate. When an enterprise participates within an enterprise network, the number of coexisting EMLs is likely to increase subsequently. The number of necessary translators would increase dramatically and it would become uneasy to keep a global consistency.

A solution to significantly decrease the number of necessary translator and to keep a global consistency between the shared knowledge is to define an unifier intermediate language that allows to represent a unique, consistent and modular version of the shared knowledge. Such a language (depicted on Figure 2.3) is usually called UEML (Unified Enterprise Modelling Language). It is not a substitute to the other MLs but it provides a useful help for the translation and the integration

of models. UEML would enable enterprises to keep their own EMLs while cooperating. This is the solution we choose to put into practice in the rest of the work. Chapter 3 will give an overview of what has already been done with this strategy.

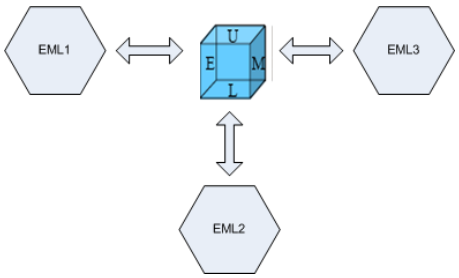


Figure 2.3: EM translation by a UEML (adapted from [BPP04]).

2.3 GRL

At this stage, we will present the Goal-oriented Requirements Language [Yu97, ITU03] which is used to provide an example throughout our work. The presentation is inspired by [DHP05, MHO06]. GRL is a EML but is not usually presented as an EML because it originates from the IS and requirements engineering communities. However it really helps in EM and especially in goal-oriented modelling and reasoning about requirements, especially for dealing with non-functional requirements. This kind of modelling is different from the detailed specification of what is to be done. Here, the modeler is primarily concerned with exposing "why" certain choices for behaviour and/or structure were made or constraints introduced.

GRL provides constructs for expressing various types of concepts that appear during the requirements process. It has four main categories of concepts: *intentional elements, links, actors* and *non-intentional elements*. The constructs of these categories are described in [Yu97] and summarized in Table 2.1.

Figure 2.4 presents a meeting scheduler example adapted from [Yu97]. It consider a computer-based meeting scheduler for supporting the setting up of meetings. The requirements might state that for each meeting request, the meeting scheduler should try to determine a meeting date and location so that most of the intended participants will participate effectively. The system would find dates and locations that are as convenient as possible. The meeting initiator would ask all potential participants for information about their availability to meet during a date range, based on their personal agendas. This includes an exclusion set of dates on which a participant cannot attend the meeting and a preference set dates preferred by the participants for the meeting. The meeting scheduler comes up with a proposed date. The date must not be one of the exclusion dates, and should ideally belong to as many preference sets as possible. Participants would agree to a meeting date once an acceptable date has been found.

Figure 2.4, shows the three actors and the dependencies between their activities. The main task of the `meeting initiator` is to organize a meeting. This task is decomposed in three sub-goals: two softGoals (`Quick and low effort`) and one goal (`Meeting be scheduled`). The `meeting initiator` have two alternatives to achieve these goals: `schedule meeting` or `let scheduler schedule meeting`. The former contributes in a negative way to the two soft goals while the latter contributes positively to them. The task `let scheduler schedule meeting` is dependent on the goal `meeting be scheduled` which is shared by the `meeting initiator` and the `meeting scheduler`. This goal (`meeting be scheduled`) is also dependent on the task `schedule meeting` of the `meeting scheduler`.

Table 2.1: GRL constructs summary.

Cat	Construct	Definition	Presentation
Intentional elements	Goal	A condition or state of affairs that the stakeholders would like to achieve	
	SoftGoal	Goal for which there are no clear-cut criteria to determine whether this condition is achieved	
	Task	describes a particular way of doing something	
	Ressource	an entity for which the main concern is whether it is available	
	Belief	express design rationale	
Links	means-ends	describe how goals are in fact achieved, typically through tasks	
	Decomposition	defines the subcomponents of a task, typically (but not limited to) the subgoals that must be accomplished	
	Contribution	describes the impact that one element has on another by design	
	Correlation	the same as a contribution link except that the contribution is not an explicit desire, but is a side-effect	
	Dependency	describes an intentional relationship between two actors	
Actor	Actor	holders of intentions	

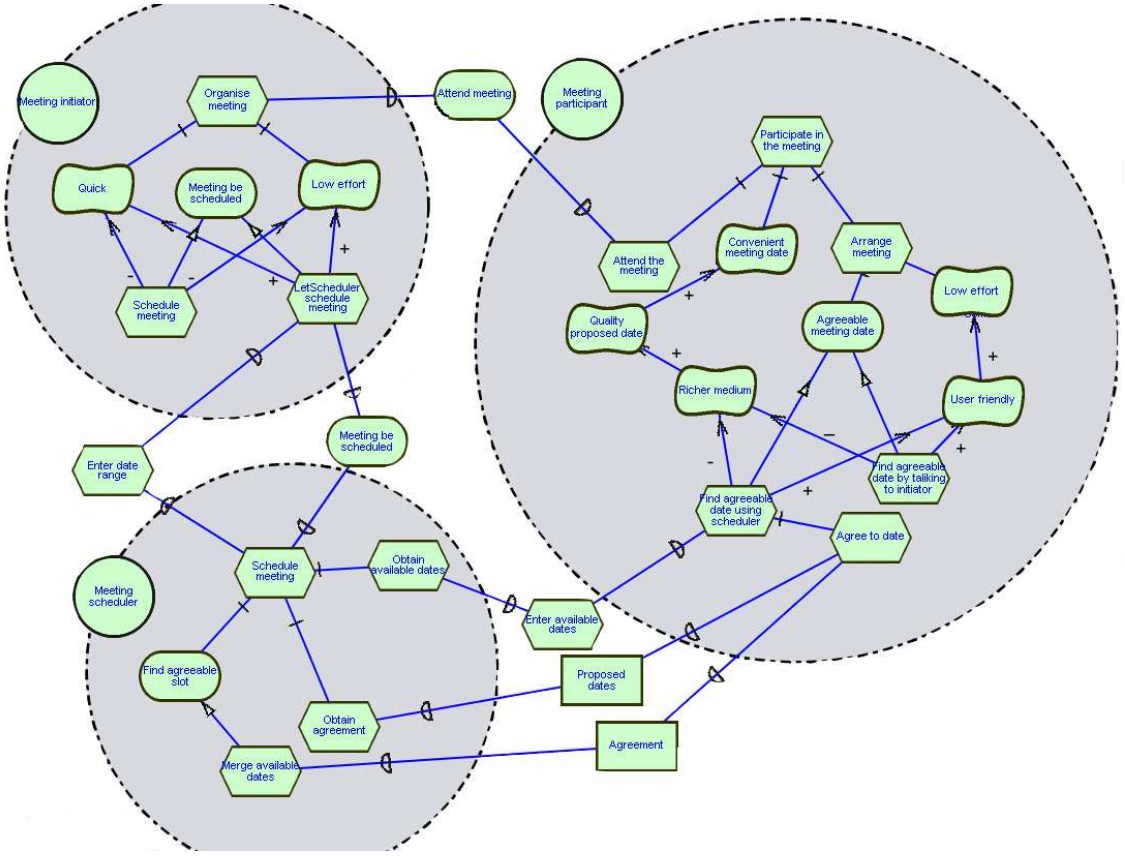


Figure 2.4: GRL Example (from [Yu97]).

2.3.1 Intentional elements

The *intentional elements* in GRL are *goal, softgoal, task, resource* and *belief*. They are intentional because they are used for models that allow answering questions such as why particular behaviors, informational and structural aspects were chosen to be included in the system requirements, what alternatives were considered, what criteria were used to deliberate among alternative options, and what were the reasons for choosing one alternative rather than the other. A *goal* is a condition or state of affairs that the stakeholders would like to achieve. As a goal, a *softgoal* is a condition that the stakeholder wants to achieve, but there are no clear-cut criteria to determine whether this condition is achieved or not (e.g., Quick, Low effort). A *task* describes a particular way of doing something (e.g., Organise meeting or Schedule meeting). A *resource* is an entity for which the main concern is whether it is available (e.g., Proposed dates). A *belief* is used to express design rationale.

2.3.2 Links

The *Link* category includes *means-ends, decomposition, contribution, correlation* and *dependency*. The *Means-ends* links are used to describe how goals are in fact achieved, typically through tasks. Each task provided is an alternative means for achieving the goal. Normally, each task would have different types of impacts on softgoals, which would serve as criteria for evaluating and choosing among each task alternative. The *decomposition* link defines the subcomponents of a task, typically (but not limited to) the subgoals that must be accomplished. The *contribution* link describes the impact that one element has on another by design (i.e., how softgoals, task, believes,

or links contribute to others). A contribution is an effect that is a primary desire during modelling. The *correlation* allows to express knowledge about interactions between intentional elements in different categories, and to encode such knowledge. A correlation link is the same as a contribution link except that the contribution is not an explicit desire, but is a side-effect. The *Dependency* link describes an intentional relationship between two actors (i.e., one actor (Depender) depends on another actor (Dependee) for something (Dependum)).

2.3.3 Actors

Actors are holders of intentions and characterize active entities (e.g., `Meeting initiator, Meeting participant`), who want goals to be achieved, tasks to be performed, resources to be available and softgoal to be "satisfied". Graphically, an actor may optionally have a boundary, with intentional elements inside.

2.3.4 Non-intentional elements

Non-intentional element is a reference to non-construct instance outside a GRL model. The main concern of these clauses is not to capture the syntax and semantics of the external model but to serve as references to the external model only. Non-intentional elements definition is used to navigate through the non-intentional model.

2.4 Summary

In this chapter, we have described what an EML is and how and why it is used in enterprises. Different EMLs have been used for specific needs even within a company. Therefore, interoperation between these EMLs has become an important objective. There are two main reasons leading to the need of interoperability. Companies need several EMLs to model their own activities and companies need to exchange their knowledge while cooperating with others. GRL (Goal-oriented Requirements Language), an example of EML which will be used throughout this work, was presented. In the next chapter we will introduce and explain how UEML provides a solution to interoperability.

Chapter 3

Overview of UEML

The previous chapter explained how companies use different EMLs and tools to reach a better integration inside the organisation. The integration efficiency depends on the way the modelling languages can cooperate. In order to integrate them a solution is to use a Unified Enterprise Modelling Language (UEML). This solution is the one that enable one to use the least necessary number of translator.

In this chapter we explain what is UEML and what are its goals and method. There are two different versions of UEML (1.0 and 2.0). Both have a common overall goal: support enterprise model exchange (integration, translation and transformation) and global consistency between evolving enterprise models. However, as we will see, they have different specific goals and hence adopt different approaches.

3.1 UEML 1.0

UEML (Unified Enterprise Modelling Language) was a Themantic Network Project (IST200134229) financed by European Union (EU). The project started on March 1st 2002 and ended May 30th 2003. The aim of UEML 1.0 was to create a working group involved in the 6th Framework Programme to develop core UEML. We will first present its purpose, then how it works and finally the limitations of this first version.

3.1.1 Purpose

UEML is the acronym for Unified Enterprise Modelling Language. It can be defined by interpreting the acronym as follows:

> "***Unified*** and shared linguistic context for ***Enterprise Modelling*** supporting all the needed tasks for representing and utilizing enterprise knowledge through a ***Language*** with well defined syntax, and possibly, semantics." [Ber03]

The UEML project is reported in [PD02] as being set up in an attempt to contribute in resolving the problems of having multiple EMLs. The long term objective of UEML is the definition of a Unified Enterprise Modelling Language, which would serve as an interlingua between EM tools. This language will:

- provide the business community with a common visual, template based language to be used on top of most commercial EM and workflow software tools;

- provide standardised mechanisms for sharing knowledge models and exchanging enterprise models among projects, overcoming tool dependencies;
- support the implementation of open and evolutionary enterprise model repositories to leverage enterprise knowledge engineering services and capabilities.

UEML is thus an intermediate unifying language that allows to represent a unique, coherent and modular vision of the shared knowledge. It is not a new modelling language that would replace the existing ones but it helps to integrate and to translate different models. UEML is therefore a pivot language that would allow enterprises to continue to use their own EMLs. Moreover, it should be extendable. The main results of the UEML project are:

- a state of the art description of Enterprise Modelling;
- a collection and categorization of requirements for the UEML language;
- an initial set of core UEML constructs presented in a meta-model (Figure 3.2);
- a demo showing the core constructs used in 3 different modelling tools and a demo of XML exchange between them;
- a survey concerning Enterprise Modelling tools.

3.1.2 How it works

The main goal of the first version of UEML was to show the feasibility of the UEML definition with respect to the objectives. The project was achieved with three EMLs: IEM (Fraunhofer) [MJ99], EEML (Computas) [EEM] and GRAI (Graisoft SA) [DVC98, DVZC92].

It produced a state of the art and a collection and categorization of requirements for the UEML language related in [KBB03]. The integration problems were emphasized and solutions were proposed to finally give a strategy for UEML 1.0. These problems and their resolutions are explained in the following subsections. We show the 10 most important requirements according to [KBB03] in Table 3.1.

Integration problems

When translating knowledge represented in one EML to another, it is important to define some commonality between constructs belonging to each language. This is why UEML comprise constructs common to the most representative existing EMLs. In the integration process the following problems ([BPP04] and [BAO04]) have to be tackled:

- differences in the (abstract) syntax: even in a unique business domain, EMLs differ syntactically. For instance, two companies could respectively use IEM and EEML.
- coverage and expressivity of the Enterprise Modelling Languages: Enterprise Modelling Languages have different focuses and purposes.
- differences in semantics of similar constructs: sometimes enterprises use the same constructs but they associate distinct meanings.

UEML handles these integration problems within the language dimension, in the following ways:

Differences in the (abstract) syntax: UEML is founded in the agreement concerning the structure (i.e. the Constructs) of the knowledge (i.e. the Concepts underlying the Constructs)

Table 3.1: Top 10 general Methodology requirements names and descriptions for UEML 1.0 (from [KBB03]).

Nr	**Name**	**Description of requirements**
1	Low resource demands	UEML should be easy to use and should demand a low level of resources for the definition of models
2	Model hard and soft aspects of human and team engineering	UEML should provide capabilities to model hard and soft aspects of human participative and resource systems in support of team engineering
3	Invariant and unique behavioural semantic	UEML need invariant and unique behavioural semantic. This is important for stable exchange and common understanding of EMs
4	Ability to capture business rules	UEML should help to capture and explain business rules.
5	Fast, easy and safe cooperation between enterprises	UEML should provide methodologies for fast, easy and safe cooperation's by using EMs (hiding of special informations)
6	Management of increasing degree of (un)certainty in enterprise structures	UEML should provide capabilities to manage the different degrees of certainty in an enterprise
7	Separately knowledge, information and data representation, and manage them with their own rules	UEML should represent the knowledge, information and data with their own rules and don't mix them
8	Usable for people who are not familiar with graphical modelling languages.	UEML should be usable for model-weak people.
9	Human cognitive processes management	UEML should help to manage the human cognitive processes
10	Identify critical enterprise processes for merging and acquisition	UEML should provide methodologies for the identification of the most critical processes and objects for the merging and acquisition of enterprises (e.g., controlling, logistic) - because in case of enterprise merging and acquisition there is a lot of time pressure. Normally both enterprises do not have EM but with EMs the quality of merging will be improved and the time expenses will be reduced. The modelling of all the enterprise objects and processes takes too much time

being exchanged or translated. This (abstract) syntax is expressed through a class model and specifically (just for convenience) a UML class model.

Coverage and expressivity of the Enterprise Modelling Languages: the decision to be taken is related to what kind, and to which extent knowledge should be shared by using UEML; in this case, the following equation has been defined:

$$UEML = CommonConcepts + (someof)NonCommonConcepts$$

This equation well represents the UEML as a federator containing what it is shared among various languages.

Differences in semantics of "similar" constructs: it is the most difficult type of problem and it is related to the definition of common concepts. A part of the complexity is due to the fact that the underlying concepts of most EMLs constructs are provided by text expressed in natural language which is ambiguous. Using these texts for finding semantic relationships between constructs belonging to distinct languages is thus very difficult and quite subjective. However, database integration suggests to use examples to cope with that kind of problem. The purpose is to identify and verify clearly the "intentional relationships between concepts" by using the evidence of available examples of such concepts. In practice, if two tables of different schemata allow to represent the same instances, we can suppose that those tables are equivalent. As showed in Table 3.2 an analogy can be done between languages and databases. Modelling languages artifacts play the role of database's instances, ML's model plays the role of the database and, finally, the ML's meta-model plays the role of the database schema. We can thus apply the database technique in order to compare MLs meta-models.

Therefore, in the UEML project, a complex scenario was defined. This scenario plays the role of databases and instances, by using the three modelling languages cited above (IEM, EEML and GRAI). Meta-models for each of these languages were also built. Afterwards, these meta-models were compared based on models and model artifacts, part of the scenario. Based on such comparisons, in the simpler case of two constructs with their underlying concepts C1 and C2 respectively in EML1 and EML2, the following types of semantic correspondences were found:

1. $C1 = C2$
2. $C1 \supseteq C2$
3. $C1 \subseteq C2$
4. $C1 \cap C2 \neq \emptyset$.

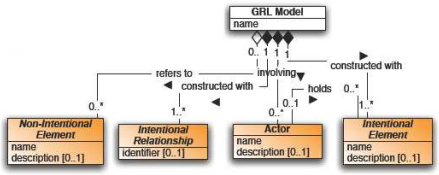
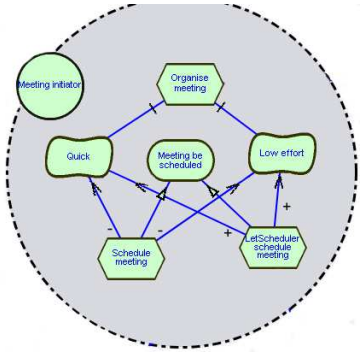
For instance, the first semantic correspondence (1) can be paraphrased as: by looking into the available models, represented in EML1 and EML2 respectively, the model artifacts represented through the concept C1 are also represented by C2 and vice-versa. In the general case, a common concept C can be introduced if $C1 \cap C2 \neq \emptyset$.

Strategy for UEML 1.0

In this first version of UEML the results in term of a common meta-model were restricted to the three languages: IEM, EEML and GRAI. The following strategy (also shown in Figure 3.1) was applied to define UEML 1.0 [BPP04]:

1. A scenario is defined and modeled in each language. This task has to be achieved by modelling experts of the languages. But of course, as experts are humans, a part of subjectivity is still possible.

Table 3.2: Analogies between modelling languages and databases (adapted from [BAO04]).

Level	Database Glossary	Modelling Language Glossary
1	Database Schema `Example:` 	Meta-model `Example:` 
2	Database (a set of related instances) `Example:` Album Title | Number | Serie | Author Le Barbare | 27 | Thorgal | Van Hamme Kriss de Valn | 28 | Thorgal | Van Hamme Author firstname | surname | nickname Georges | Rémi | Hergé Pierre | Culliford | Peyo Jean | Van Hamme | Serie Name | authfirstname | authname Thorgal | Jean | Van Hamme	Model (a set of related model artefacts) `Example:` 

3	Data (instances)
	`Example:`
	`Thorgal / Van Hamme / Le Barbare / Kriss de Valnor / ...`
		Model artifacts
		`Example:`
		`Meeting initiator / Organize meeting / Quick / Low effort / ...`

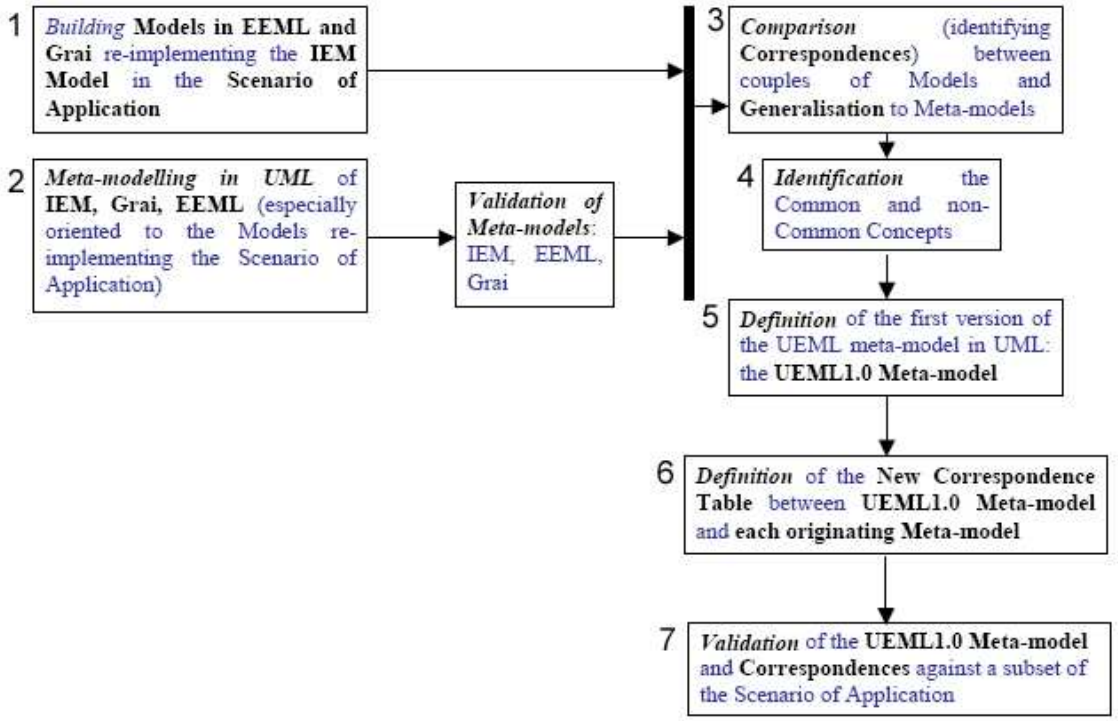


Figure 3.1: Strategy for UEML.

2. Each language's meta-model is defined in a UML class Diagram. Those meta-models represent the languages abstract syntax by a set of classes and relationships between them. They are checked by UML experts.
3. Semantic correspondences are established between the three languages.
4. Common concepts and some non-common concepts are identified.
5. A version of UEML can be made with the common concepts identified in the previous step. Some non-common concepts are also added.
6. A final version of semantic correspondences between the languages and the UEML meta-model is defined.
7. A final check of the semantic correspondences between the languages and the UEML meta-model is done (for instance with new scenarios).

The UEML meta-model of figure 3.2 is the result of this strategy used with the three languages. Table 3.3 is a summary of semantic correspondences between the three languages and UEML 1.0 obtained as a result of step 6.

3.1.3 Discussions

The benefits and the problems of UEML 1.0 given by [Ber05a] can be summarized as follows:

Several underlying **benefits** that can be gained by the UEML 1.0 approach can be pointed out:

- *Practical*, because the method proposes at least one suitable way to map one language onto another one;

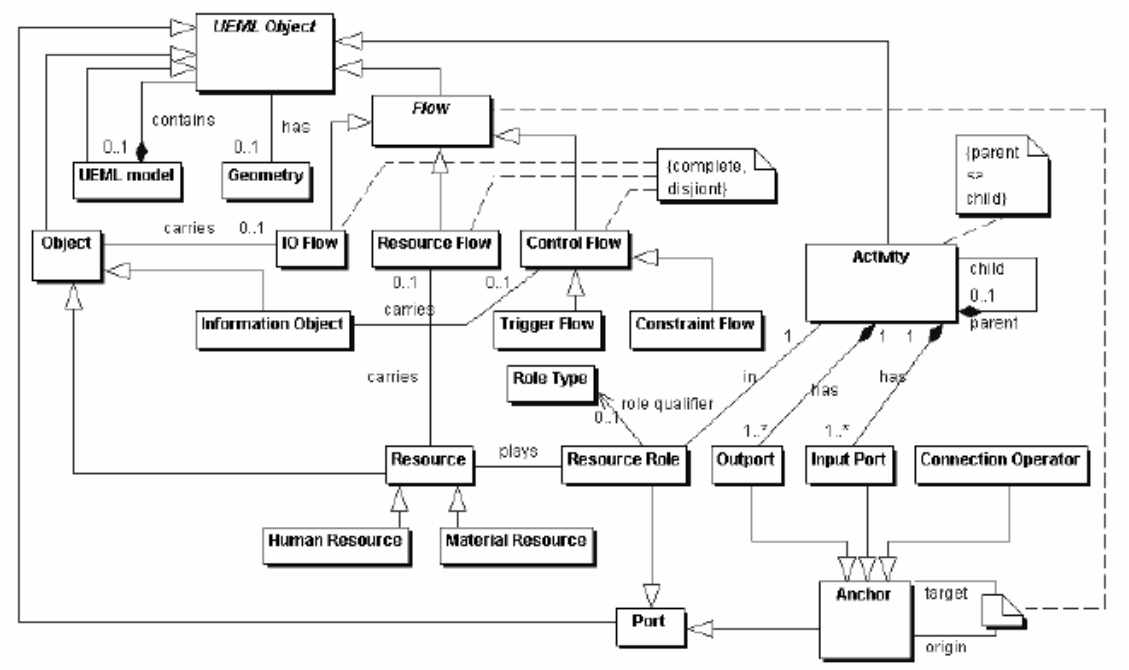


Figure 3.2: UEML 1.0 meta-model (from [BPP04]).

Table 3.3: Semantic correspondences between IEM, EEML, GRAI and UEML 1.0 (from [BPP04]).

Common concept	GRAI	IEM	EEML
ACTIVITY	Extended activity	Action state	Task
ROLE	Not explicit	IEM Object state	Role
RESOURCE	Resource	Resource class	Resource
INPUT / OUTPUT / FLOW	Input / Flow Ouput/ Flow	Successor / ProcessElement	Flow (with is control flow = false)
CONSTRAINT FLOW	Control / Flow (with trigger=false)	No direct equivalence	Flow (with is control flow = false and Role)
CONTROL FLOW	Control / Flow (with trigger=true)	ControlSuccesor / ProcessElement	Flow (with is control flow = true)
RESOURCE FLOW	Resource / Flow (with trigger = false)	ResourceSuccessor / Resource State	Role (with Task)
CONNECTION OPERATOR	Logical operator	Connection Element State	DecisionPoint (not (Inport or Outport))
PORT	Connector	Port	Decisionpoint (Inport or Outport)

- *Conceptual*, because the model artefacts represented in a language and further represented by UEML constructs, are understandable in terms of what they are intended to represent (because the correspondences tend to be basic correspondences);
- *Potential*, because, on the top of the simple exchanges of models that can be realised by using only the identified correspondences, more complex exchanges may also be realised iff a mapping language is available for representing mappings over meta model artefacts.
- *Architectural*, because the approach allows to implement an architecture in which it is possible to provide a uniform interface for accessing models represented in several languages;
- *Methodological*, because new relationships between UEML constructs, not available between language constructs (because distinct languages are not related) can be identified (and new methods and methodologies can be developed).

The following **problems** are also pointed out:

- The proposed approach seems difficult to be generalized, to be suitable for managing situations of potential inconsistencies of correspondences (e.g. a construct in a language can represent several constructs in another language), and to be independent from the models used to find out the correspondences and from the modellers building these models.
- Between two fully formalized languages (even between a language and itself) there are several possible correspondences: the problem is still how to identify correspondences that might be basic correspondences.
- The advocated specific mappings that realise complex exchanges can be represented if a specific mapping language is available. However, the UEML 1.0 approach does not help to formally prove (correctness) properties of these exchanges.
- It is uneasy to know how UEML reached its objectives because the requirements are unclear and fuzzy.

3.2 UEML 2.0

The second version of UEML is currently being developed (from 2003 to 2007) in the INTEROP Network of Excellence (NoE) (IST-1-508011). INTEROP aims to create the conditions of an innovative and competitive research in the domain of Interoperability for Enterprise Applications and Software. The integration will be achieved by the end of the 3-year project duration. Meanwhile, INTEROP spreading of excellence activities should ensure the fertilisation of the largest research community as well as IT providers and users, to provide a durable Virtual Lab on Interoperability beyond the EU-funded period. The UEML work in INTEROP has three main activities:

1. determining requirements for UEML;
2. selecting languages to incorporate into UEML;
3. describing the modelling constructs that are chosen as part of UEML.

As for UEML 1.0, we first present its purpose and then how it works.

3.2.1 Purpose

[Opd06] presents UEML 2.0 as follows:

> The Unified Enterprise Modelling Language (UEML) is an ongoing effort to develop an intermediate language for modelling enterprises and related domains, such as IS. Being an intermediate language, the aim of UEML is not to propose new modelling constructs or new visual model presentations. Instead, the aim is to integrate existing modelling languages in a structured and cohesive way. In the longer term, the UEML can thus potentially support for comparison, consistency checking, update reflection, view synchronization and, eventually, model-to-model translation across modelling language boundaries.

The following principles ([Ber05b]) were taken into account to define the UEML approach:

1. **Integrative approach**: The UEML should not propose a new language. Instead, the UEML should be built on existing industrial and experimental languages and try to integrate them. The approach allows to select only part of a language and maintains clear relationships between the existing languages and the UEML core language because constructs belonging to existing languages are directly represented.

2. **Extendable, tailorable**: It should be possible to incorporate new modelling languages into the UEML when needed and to locally tailor the UEML to fit specific organisational standards. In other words, UEML should promote an open-ended approach. The approach recognises that languages are difficult to be compared without a context of application and provides a very general way to model modelling languages.

3. **Standardised and template-based**: All diagram type definitions and modelling construct definitions in the UEML should be in a common standardised format (see the text-based template in Chapter 4)

4. **Separate presentation from content**: The UEML should have clearly separated meta-models for presentation (lexical, syntax and related issues) and content (semantic issues). On the presentation side there should be one meta-model for each included type of diagram. On the content side there should be a common meta-model that integrates the semantic domain of the various languages. This meta-model would thus represent semantic and not syntax as it usually does (see Chapter 6). Each presentation meta-model should of course be linked (or mapped onto) the content meta-model.

5. **Structured approach to organise and manage the common meta-model**: As more languages are added to the UEML, a major risk is that the common meta-model becomes messy. Providing principles for extending and evolving the content meta-model is therefore important. Additionally, the approach suggests the reuse of object classes and properties included in the semantic domain; this constitutes the base for defining a UEML core language.

6. **Industrial languages**: The UEML should give priority to industrial languages to make sure the work is practically relevant.

3.2.2 How it works

In this section we explain the mechanisms of the UEML 2.0 approach. We first present the criteria to choose the languages to study, then the general approach and the three parts of the template and finally the meta-meta-model.

Language selection

In order to select which languages should be part of UEML 2.0, a method called "The Extended Quality Framework" has been considered. It is composed of a list of criteria based on a quality assessment which is made through six areas of appropriateness [OA05, KLS]:

- *Domain appropriateness.* Ideally, the conceptual basis (i.e. the set of concepts embedded by the language) should be powerful enough to express anything in the domain and nothing else. We pay attention to construct deficit (when the conceptual basis is not be powerful enough to express anything in the domain) and to construct excess (when the conceptual basis enables one to express things that are not in the domain).
- *Knowledge appropriateness.* The conceptual basis should correspond as much as possible to the way people perceive reality. But it depends on the persons and the persons knowledge is not static. The external representation of the different phenomena should be intuitive.
- *Knowledge externalization appropriateness.* The goal is that there is no statement in the explicit knowledge of the participant that cannot be expressed in the language. This focuses on how relevant knowledge can be articulated in the modelling language.
- *Comprehensibility appropriateness.* Participants should understand all the possible statements of the language. The following requirements should be fulfilled:
  - the number of concepts should be reasonable,
  - the concepts should be general,
  - the concepts can be composed (related statements can be grouped in a natural way),
  - the language must be flexible in precision,
  - the language must be formal and unambiguous,
  - the language must have constructs for modelling non-detailed knowledge (that are also formal).
- *Technical actor interpretation appropriateness.* The language itself should lend to automatic reasoning.
- *Organizational appropriateness.* How appropriate is the language for the organization using it, taking into account standardization on technology, tools and modelling methods within the organization. In this case, it should be noted that the ORGANIZATION is INTEROP itself. However, not all the criteria related to the organizational appropriateness should be taken into account, because the objective of INTEROP is not to apply languages for modelling but to provide new solutions. It can be debatable if a language should not be part of a UEML just because in INTEROP there is no skill about it. In fact, INTEROP should also be able to acquire skills on specific languages.

Strategy

We present on Figure 3.3 the UEML 2.0 general strategy. This strategy consists in dividing the languages in different constructs one has to analyse. By using the UEML template, these constructs will reveal the classes, properties, states and transformations (see below) they represent in order to build the UEML ontology. This ontology will enable one to find several categories of correspondences and to generate the UEMLCore.

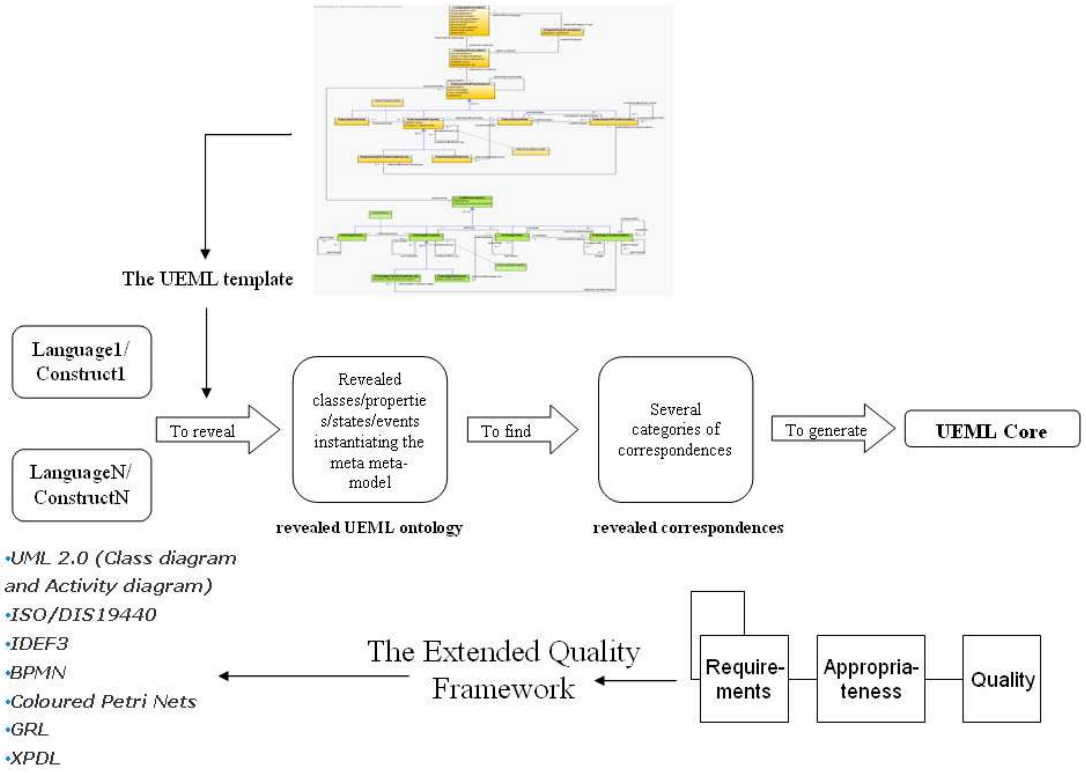


Figure 3.3: UEML 2.0 general strategy (adapted from [Ber06]).

General approach

The UEML 2.0 approach is described in [Opd06] in the following seven features:

1. A construct description is created in a structured and cohesive manner for each modelling construct that is to be incorporated into UEML (Figure 3.4).

2. The construct description has both a presentation part, dealing with the visual presentation of the modelling construct ("lexeme", "syntax" and some "pragmatics"), and a representation part, accounting for which enterprise phenomena the construct is intended to represent ("reference", an aspect of "semantics").

3. The representation part uses referential decomposition to break each modelling construct into its ontologically atomic parts, defined as a part that maps one-to-one with an ontology concept, e.g., a particular class of things in the enterprise (ProductionEquipment), a particular type of property or relationship (having a responsibility), a particular state (being idle) or a particular transformation (acquiring a new responsibility). The representation part and the underlying idea of referential decompositon are based on [OHS04, OHS05].

4. The ontology concepts are maintained in a common ontology, which grows incrementally as more modelling constructs are incorporated into the UEML. The common ontology is based on Bunges ontological model [Bun77, Bun79] and the Bunge-Wand-Weber representation model (the BWW model, [WW88, WW95, WW93]) of IS. The initial classes, properties, states and transformations are drawn from Bunge 's ontology and the BWW-model, but the common ontology also grows dynamically as more specific classes, properties, states and transformations are included (Figure 3.5).

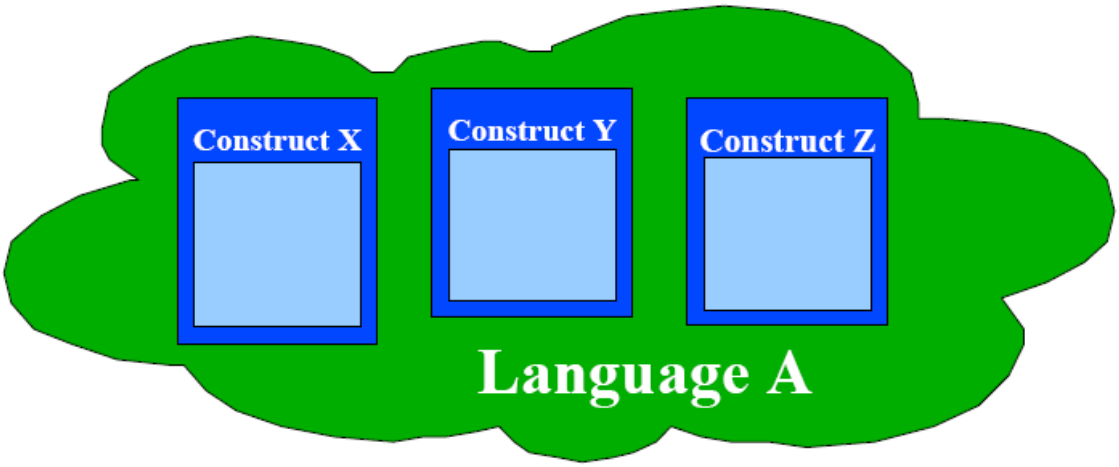


Figure 3.4: Each modelling construct of the language is described separately (from [Opd06]).

5. The common ontology is hierarchically organized. Ontology classes are organized using generalisation / specialisation (ProductionEquipment is subclass of Equipment) and using aggregation / decomposition (Equipment is composed of EquipmentParts). Properties are organized using property precedence (being human precedes having a responsibility). States are similarly organized using super-/substate relationships and transformation using super-/subTransformation relationships (also shown in Figure 3.5).

6. All the modelling constructs in the UEML are thereby interrelated at the most detailed level possible via the common ontology: if two modelling constructs are identical, they will map onto the exact same ontology concepts. If two modelling constructs do not overlap at all, they will map onto completely distinct ontology concepts, i.e., onto concepts which are not even hierarchically related. The third case is likely to be most common, where two modelling constructs map onto some identical ontology concepts, some ontology concepts that are hierarchically related and some ontology concepts that are completely distinct. But in all cases, the hierarchically organized common ontology makes it possible to determine the exact representational ("semantic") relationship between any pair or group of modelling constructs (Figure 3.6).

7. A meta-meta-model (Figures 6.1 and 6.2 explained further in Chapter 6) is provided to account for the representation part of the UEML approach (it does not yet account for the presentation part), based on earlier work: [OHS04, OHS05]. It is called a meta-meta-model because it is a model of how to model languages and because models of languages are called meta-models. The top layer of the meta-meta-model deals with modelling languages, their diagram types and their modelling constructs. The middle and bottom layers deal, respectively, with individual construct descriptions and with the common ontology.

The three parts of the template

In order to follow the general approach, a text-based template has been set up. This text-based template is more deeply explained in Chapter 4. Analyzing a modelling construct is done by filling it in (see Section 4.1). As explained in ([Opd05]) the template is made of three parts: preamble, presentation and representation.

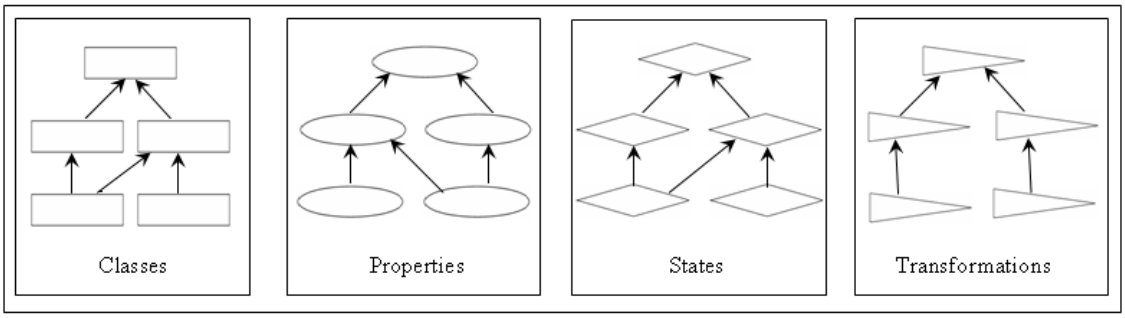


Figure 3.5: Organisation of the common ontology (adapted from [Opd06]).

The Preamble section corresponds with the upper layer of Figure 6.1. It deals with general issues about the modelling construct such as the name, the version or the relationships to other modelling constructs.

The Presentation section of the template is not taken into account in Figures 6.1 and 6.2. "It describes the visual presentation of the modelling construct. Presentation issues include lexical information (such as icons, line styles), syntax (how this and other modelling constructs connect in diagrams and repositories) and some pragmatics (in particular layout conventions). The Presentation section of the template has been kept informal at this stage, because efforts have been focussed on the following Representation section, which is believed to be more difficult. Earlier versions of the template used the term "Syntax" to describe this section. It has been renamed because it is not only about syntax, but also about lexicals and pragmatics. The Presentation section deals with lexical, syntax and practical informations" [Opd06].

The Representation section deals with the semantic aspects of the constructs. "It corresponds to the top three classes of Figure 6.1, in particular to the ConstructDescription class. It describes that instantiation level, classes, properties and kinds of dynamic behaviour that a modelling construct can be used to represent. Most existing modelling language definitions describe semantics using text only, so the entries in this template section usually cannot be filled in without interpreting the language definition, looking between the lines to some extent, and also looking at examples of how the language is used in practice" [Opd06]. This is the most difficult and most important part of the template.

3.3 Comparison

Previously, we presented UEML 1.0 and 2.0. Within this section, we point out some differences in their objectives and approaches.

3.3.1 Goal

The two versions aim to support enterprise model exchange (integration, translation and transformation) and global consistency between evolving enterprise models. However they have different priorities. First, one of the main goals of UEML 1.0 - *to translate different models* - is clearly revised downwards in the second version. Indeed, in the second version, it becomes an eventual consequence of the approach. Secondly, the UEML 2.0 approach offers more perspectives such as consistency checking, update reflection or view synchronization.

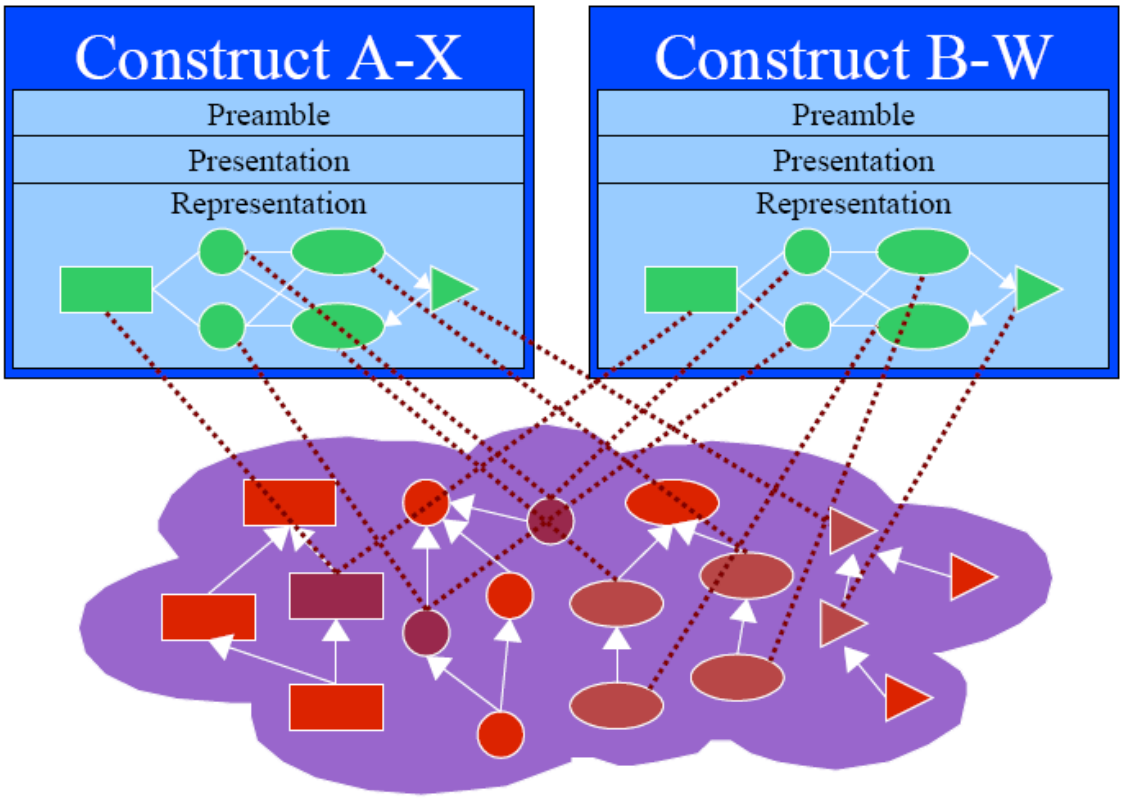


Figure 3.6: Construct referential decomposition (from [Opd06]).

3.3.2 Method

Both approaches require to represent the abstract syntax but the second version has a standardised way to do it based on the template. UEML 2.0 undertakes a very different, eventually complementary approach. Basic correspondences are not statically defined as in UEML 1.0 step 4 of the strategy (Section 3.1.2) but should be inferred according to the represented semantics (as shown on Figure 3.6).

In UEML 1.0, there is no explicit representation of semantics. Abstract syntax is defined but not the semantic. This is done is the second version with the Representation part of the template.

UEML 1.0 approach is based on three specific languages and nothing is done to include new ones. UEML 2.0 approach was though in a way that always enable to add new language analyses.

[Ber05a] gives the advantages of the UEML 2.0 approach in comparison with those of the first version: The meta-meta-model (depicted in Figures 6.1 and 6.2) introduces, formally, the notion of construct, forcing to become clear on the abstract syntax. All the benefits that have been mentioned in for UEML 1.0 (Section 3.1.3) are mostly similar for the UEML 2.0. However, the UEML 2.0 approach improves the UEML 1.0 approach:

- to guide, according to a meta-meta-model, the representation of abstract syntax, semantics and semantic domain for any enterprise modelling language;
- to infer basic correspondences between distinct languages, once their semantics and the semantic domain have been represented;

- to represent some UEML constructs and their semantics whenever the semantic domain contains enough information (i.e. whenever a significant number of relevant languages has been represented);
- As for UEML 1.0, the undertaken approach does not allow to formally prove properties of basic correspondences and more complex exchanges because the meta-meta-model is represented with a UML class diagram. However, the meta-meta-model can be represented in some formal language enabling reasoning.

3.4 Summary

In this chapter, we described the purpose of UEML and its evolution through the two versions. Of course, the approach had to adapt to this evolution. While the first version was only based on three languages, the second is more ambitious and aims to be extendable to many different languages. In the remainder of this work we will focus exclusively on the second version which is notably based on a template and a meta-meta-model we are going to explore more deeply in the following chapters.

Chapter 4

The UEML 2.0 Template

In the previous chapter we exposed the general approach of UEML 2.0. We noticed it is based on a template that helps one to analyse a language and its constructs. In this chapter, we will detail the different parts of the template and how to use it.

4.1 Generalities

In order to incorporate a language into UEML, each of its modelling constructs has to be described in a standard way following the general approach of Section 3.2.2. This approach can be achieved by using the text-based form called the template.

At the beginning of the UEML 2.0 project, the template was only text-based. The way to follow the method is to answer to the different questions one can find in a text-document called "UEML Template". As explained in [Opd05], this document is divided in 3 main parts corresponding to those of Section 3.2.2: Preamble, Presentation and Representation. This document assists one to do a complete UEML 2.0 analysis of a language.

One document has to be filled in for each construct and has to be named like this:

```
"UEML_construct_<LANGUAGE_ACRONYM>_<LANGUAGE_VERSION>_
<DIAGRAM_TYPE>_<CONSTRUCT_NAME>_<VERSION_NUMBER>."
```

where LANGUAGE_VERSION and DIAGRAM_TYPE can be left out. A filled in template can be found in Section 10 where we propose an example of a language analysis with the GRL language.

4.2 Preamble

The preamble section deals with general issues about the modelling construct such as the construct name, relations to other constructs, the diagram types and the language it belongs to. This section is composed of the following entries:

- Construct Name – The name of the construct.
- Alternative Construct Names – Other names that are sometimes used for the construct.
- Related, but distinct construct names – Names of related constructs that should not be confused.

- Related terms – Terms that are not names of modelling languages but that are nevertheless useful or necessary to understand and talk about the language.
- Language – The language the construct belongs to.
- Diagram Type – The Diagram Type(s) of the language in which the construct can be used.

4.3 Presentation

The presentation section describes the visual presentation of the modelling construct. It includes lexical information, syntax and some pragmatics. The lexical information deals with things such icons or lines style, the syntax describes how the modelling construct is connected to other constructs in the diagram and the pragmatics mainly deal with layout conventions. The presentation section is composed of the following entries:

- Icon, line, style, text – The corresponding icon of the construct if it's a node or the corresponding line style if it's an edge. It must also explain how the construct can change with the value of its attributes.
- Builds On – If the presentation of the construct is based on those of other constructs, they are named here.
- Built on by – Here are named the construct that list the construct as "Builds On".
- User-definable attributes – List of the language-defined attributes that can be defined for an instance of the construct. This is different from the Properties of the Representation section because it only has to do with *how things looks on the surface*.
- Relationships to other constructs – language-defined attributes that can be defined for an instance of the construct (relationships within the same diagram type or relationships to constructs in other diagram type of the same language).
- Layout conventions – Description of the preferred way (layout rules or conventions) to use this construct when drawing visual models.

4.4 Representation

The representation section deals with the semantic aspects of the constructs. It describes the instantiation level, classes, properties, states and transformations the modelling construct can be used to represent. The representation section is composed of the following entries:

- Instantiation level – The construct is intended to represent either classes, properties, states or transformations at the type level or at the instance level or both.
- Classes of things – This entry lists the classes of things the construct is intended to represent. Each modelling construct represents a class of things even if it is not its main purpose. Indeed, a property is implicitly the one of one or more classes of things; a state is defined in terms of properties that characterize a class; a transformation possesses pre and post states that are defined in term of properties that characterize classes.
- Properties (and relationships) – This entry lists the properties the modelling construct is intended to represent if any. It also lists the relationships it represents because they are considered as mutual properties. As for the "Class" entry, properties are sometimes represented even if it is not the main purpose of the construct. Indeed, each state is defined

in terms of properties and a transformation has pre and post states. Some properties have sub-properties.

- Behaviour – This entry states the kind of behaviour the construct is intended to represent. If the construct is only intended to represent existence, then this entry is only filled in with the word "existence". If the construct represents a state, this state has to be described with an invariant over the property roles defined in the previous entry. If the construct represents a transformation, the *from* and *to states* have to be specified. If the construct represents a process the *states transformations* in the process must be specified.
- Modality – This entry tells if the language is intended to represent regular or modal assertions. If it represents regular assertion (that is actually what they usually do) this means it represents things as they are. If it represents modal assertions (like goal in i*), it means it represents things as someone wants them to be or as someone is not permitted to do, etc.

4.5 Summary

The UEML 2.0 Template is the device used to perform the language analyses. The three sections help one to formalize the knowledge about a language in order to be able to share it. It is also foreseen that, based on this knowledge, it will be possible to devise language comparison, consistency checking, update reflection, view synchronization, model-to-model translation,... The preamble and representation sections actually rely on the meta-meta-model we will present in chapter 6. Chapter 10 will give an application of the template on the GRL language.

Chapter 5

Ontology

Before continuing the exploration of UEML 2.0 and its meta-meta-model, we have to precise the notion of ontology. While presenting this notion we will also explore the OWL language and Protégé because these technologies are used in the solutions presented later. In the first section we explain what is an ontology and how to represent one with OWL. In the second section we introduce Protégé and how it can manage ontologies.

5.1 Ontology

This section is largely inspired by [Cor, Wik].

> "Ontology is the theory of objects and their ties. The unfolding of ontology provides criteria for distinguishing various types of objects (concrete and abstract, existent and non-existent, real and ideal, independent and dependent) and their ties (relations, dependences and predication)" [Cor].

Contemporary ontology is developed from both philosophers and scientists working in Artificial Intelligence, data-bases theory and natural language processing. We may therefore distinguish ontology as conceptual analysis from ontology as technology.

In philosophy, ontology is the most fundamental branch of metaphysics. It studies being or existence and their basic categories and relationships, to determine what entities and what types of entities exist. Ontology thus has strong implications for conceptions of reality.

In computer science, an ontology is a data model that represents a domain and is used to reason about the objects in that domain and the relations between them. Ontologies are usually used as a form of knowledge representation about the world or some part of it. Ontologies generally consist of:

1. Individuals: the basic or "ground level" objects,
2. Classes: sets, collections, or types of objects,
3. Attributes: properties, features, characteristics, or parameters that objects can have and share,
4. Relations: ways that objects can be related to one another.

A domain ontology (or domain-specific ontology) models a specific domain, or part of the world. It represents the particular meanings of terms as they apply to that domain. For example

the word "card" has many different meanings. An ontology about the domain of poker would model the "playing card" meaning of the word, while an ontology about the domain of computer hardware would model the "punch card" and "video card" meanings. In Chapter 6 the BWW ontology related to the domain of IS is described.

An upper ontology (or foundation ontology) is a model of the common objects that are generally applicable across a wide range of domain ontologies. It contains a core glossary in whose terms objects in a set of domains can be described. There are several standardized upper ontologies available for use, including Dublin Core, GFO, OpenCyc/ResearchCyc, SUMO, and WordNet.

5.2 Ontology languages

[PS04] defines ontology languages as follows:

> "An ontology language is a language in which it is possible to provide information about the different kind of objects in the domain of discourse (i.e. the part of the world that is of interest). Collections of such information are called ontologies."

Usually, two global entities are distinguished inside an ontology. The first part, that has a terminological objective, defines the domain of discourse's components nature. It is more or less like defining entity types in Entity Relationship diagrams. The second part of an ontology explains the relationships between the instances of the classes defined in the terminological part. Inside an ontology, concepts are defined one related to the others and it allows to consider and to manipulate this knowledge.

5.2.1 OWL

OWL stands for Web Ontology Language. It is an Official W3C Standard built on top of RDF and written in XML. The W3C describes it in [MvH04] in those words: "The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interoperability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics. OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full."

An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or multiple distributed documents that have been combined using defined OWL mechanisms.

We briefly explain the header of the OWL file and then the most important concepts: Classes, Properties and individuals. This section is based on the W3C recommendations ([DSB+04]).

Header

An OWL file must begin by specifying two things: the namespace and the header. The namespace is a link to glossaries and the header describes the ontology domain. In order to be able to use terms in an ontology, it is necessary to indicate from which glossary these terms come from. That is why, as in all other XML documents, an ontology begins with the declaration of namespaces. After the namespaces declaration, we can write a header describing the contents of the current ontology. It is done with the *owl:Ontology* axiom. Typically it also contains the following axioms:

- **owl:imports**, that references another OWL ontology containing definitions, whose meaning is considered to be part of the meaning of the importing ontology;
- **owl:versionInfo**, giving information about this version, for example RCS/CVS keywords;
- **owl:priorVersion** that identifies a specified ontology as a prior version of the containing ontology;
- **owl:backwardCompatibleWith**, that identifies a specified ontology as a prior version of the containing ontology, and further indicates that it is backward compatible with it;
- **owl:incompatibleWith** that indicates that a specified ontology is a later version of the referenced ontology, but is not backward compatible with it;
- **owl:DeprecatedClass and owl:DeprecatedProperty** that allows an ontology to maintain backward-compatibility while phasing out an old vocabulary (thus, it only makes sense to use deprecation in combination with backward compatibility).

We now introduce the main OWL concepts that allow to represent the ontologies.

Classes

A class defines a group of individuals that are together because they have similar characteristics. Each individual is an instance of the class. Depending on the way they are constructed, classes can have a name or not. A class can be constructed by naming it (class indicator), by enumerating its individuals, by restricting the properties (the class is composed of all instances that satisfy the constraints) and by intersection, union or complementarity. Here is an example of class definition with the class indicator:

```
<owl:Class rdf:ID="Human" />
```

Class axioms

OWL contains three language constructs for combining class descriptions into class axioms:

- **rdfs:subClassOf** allows one to say that the class extension of a class description is a subset of the class extension of another class description. Every class inherits from the superClass *Thing*. A class can thus specialize other classes with the special property called *subClassOf*.
- **owl:equivalentClass** allows one to say that a class description has exactly the same class extension as another class description (i.e., both class extensions contain exactly the same set of individuals). The use of this axioms does not imply class equality. Class equality means that the classes have the same intensional meaning (denote the same concept).
- **owl:disjointWith** allows one to say that the class extension of a class description has no members in common with the class extension of another class description.

Properties

Now we know OWL classes but we still have to learn how to express facts related to these classes and their instances. This is the goal of the OWL properties. There are two different kinds of properties:

- *owl:ObjectProperties* that allow to link instances to other instances;

- *owl:DataTypeProperties* that allow to link instances to data values.

Those two classes of properties are sub-classes of the rdf-class *rdf:Property*.

The definition of the property's characteristics is done with a property axiom. In its basic form, it only declares the existence of the property:

```
<owl:ObjectProperty rdf:ID="hasParent" />
```

Nevertheless it is possible to define many other property characteristics in a property axiom. As for a mathematical function, we can restrict the domain and the range of the property. For an *ObjectProperty* we can specify the kind of classes. In the case of a *DataTypeProperty* the range can be a data type as defined is the XML schema (e.g., positiveInteger). For instance we can define the property of type "yearOfBirth"

Inheritance can also be used for the properties, exactly as for the classes.

```
<owl:Class rdf:ID="Human" />
<owl:ObjectProperty rdf:ID="isFromFamilyOf">
  <rdfs:domain rdf:resource="#Human" />
  <rdfs:range rdf:resource="#Human" />
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasBrother">
  <rdfs:subPropertyOf rdf:resource="#isFromFamilyOf" />
  <rdfs:range rdf:resource="#Human" />
   ...
</owl:ObjectProperty>
</owl:Class>
```

The property "hasBrother" is a sub-property of the property "isFromFamilyOf". It means that every entity having the property "hasBrother" of a certain value also has a property "isFromFamilyOf" of the same value.

In addition to the inheritance mechanism and the restriction of the domain and the range, other ways to add characteristics to properties exist. Some of the main characteristics are "transitivity", "symmetry", or "inverse".

Individuals

In order to define an individual, we state a fact. Two kinds of facts are distinguished: facts concerning the membership to a class and facts concerning the identity of individuals.

Facts usually concern the membership declaration of an individual to a class and the properties values of this individual. A fact is expressed like this:

```
<Human rdf:ID="Nicolas">
  <hasFather rdf:resource="#Gilles" />
  <hasBrother rdf:resource="#Sebastien" />
</Human>
```

This fact states the existence of a human called "Nicolas". This human has a father called "Gilles" and a brother called "Sebastien". It is also possible to instantiate an anonymous individual by leaving out its identifier.

Problems could occur with the names we give to individuals. For instance, one individual could be called in two different ways. In order to avoid that kind of difficulties, OWL offers mechanisms

like: *owl:sameAs, owl:differentFrom* or *owl:AllDifferent*. The following example illustrates the how to express class equality:

```
<rdf:Description rdf:about="#Robert">
  <owl:sameAs rdf:resource="#Bob" />
</rdf:Description>
```

5.3 Protégé

Protégé [Pro] is a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies. At its core, Protégé implements a rich set of knowledge-modelling structures and actions that support the creation, visualization, and manipulation of ontologies in various representation formats. Protégé can be customized to provide domain-friendly support for creating knowledge models and entering data. Further, Protégé can be extended by way of a plug-in architecture and a Java-based Application Programming Interface (API) for building knowledge-based tools and applications. The Protégé platform supports two main ways of modelling ontologies:

- The Protégé-Frames editor enables users to build and populate ontologies that are frame-based, in accordance with the Open Knowledge Base Connectivity protocol (OKBC). In this model, an ontology consists of a set of classes organized in a subsumption hierarchy to represent a domain's salient concept, a set of slots associated to classes to describe their properties and relationships, and a set of instances of those classes – individual exemplars of the concepts that hold specific values for their properties.
- The Protégé-OWL editor enables users to build ontologies for the Semantic Web, in particular in the W3C's Web Ontology Language (OWL).

5.3.1 Protégé-OWL

The Protégé-OWL editor is an extension of Protégé that supports the Web Ontology Language (OWL). The Protégé-OWL editor enables users to:

- load and save OWL and RDF ontologies;
- edit and visualize classes, properties and SWRL (a Semantic Web Rule Language combining OWL and RuleML);
- define logical class characteristics as OWL expressions;
- execute reasoners such as description logic classifiers;
- edit OWL individuals for Semantic Web markup.

Protégé-OWL's flexible architecture makes it easy to configure and extend the tool. Protégé-OWL is tightly integrated with Jena (a Java API for OWL) and has an open-source Java API for the development of custom-tailored user interface components or arbitrary Semantic Web services.

Ontology management

Managing an ontology is done logically without the need to know the OWL syntax at all. Creating a class or a property is really simplified thanks to the graphical interfaces. One just needs to fill in a form from the "class" or "property" tab we can see in Figure 5.1. Those classes and properties

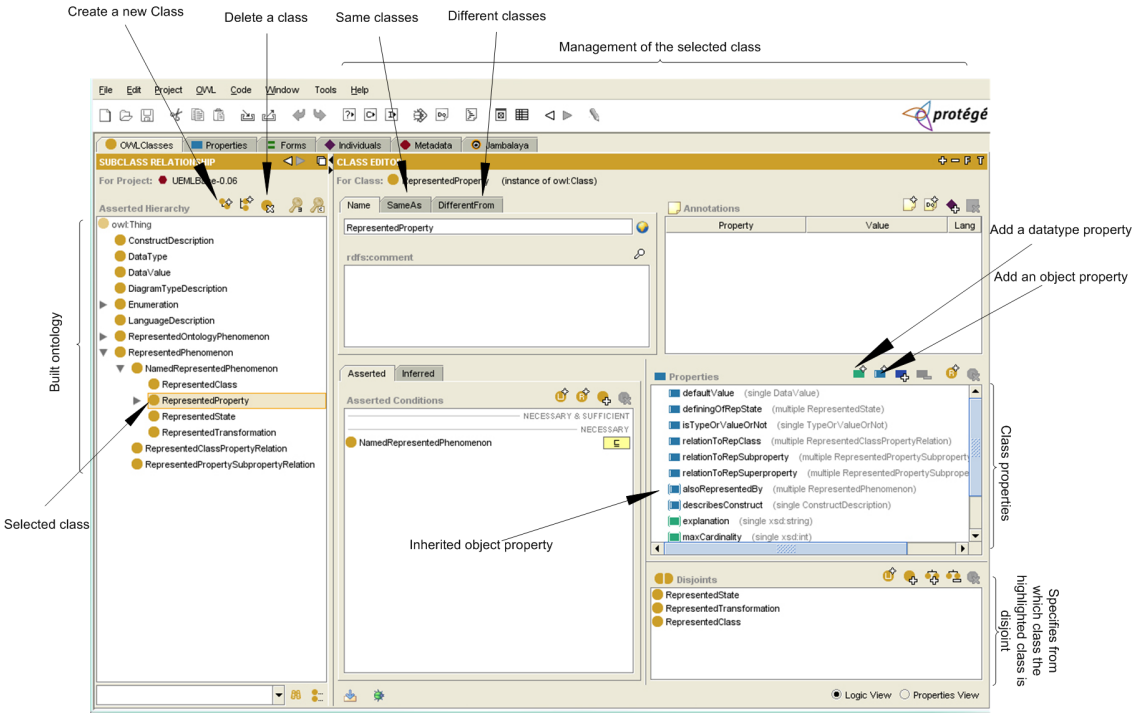


Figure 5.1: The Protégé "class" tab.

can be updated by simply changing the values of the different fields. In order to easily populate an ontology, adapted forms can be created. Those forms (Figure 5.2) are similar to the standard ones that create classes and properties. Individuals are then created like classes and properties. They can be visualized thanks to the "individuals" tab we can see on Figure 5.3 where their properties can also be changed.

5.4 Summary

In this chapter we described what an ontology is and how to express it using OWL. We also discussed how Protégé can facilitate the management of an ontology. Those tools will be used for managing knowledge about EMLs as we will see in Chapter 7. In Chapter 6 we will present the UEML 2.0 meta-meta-model.

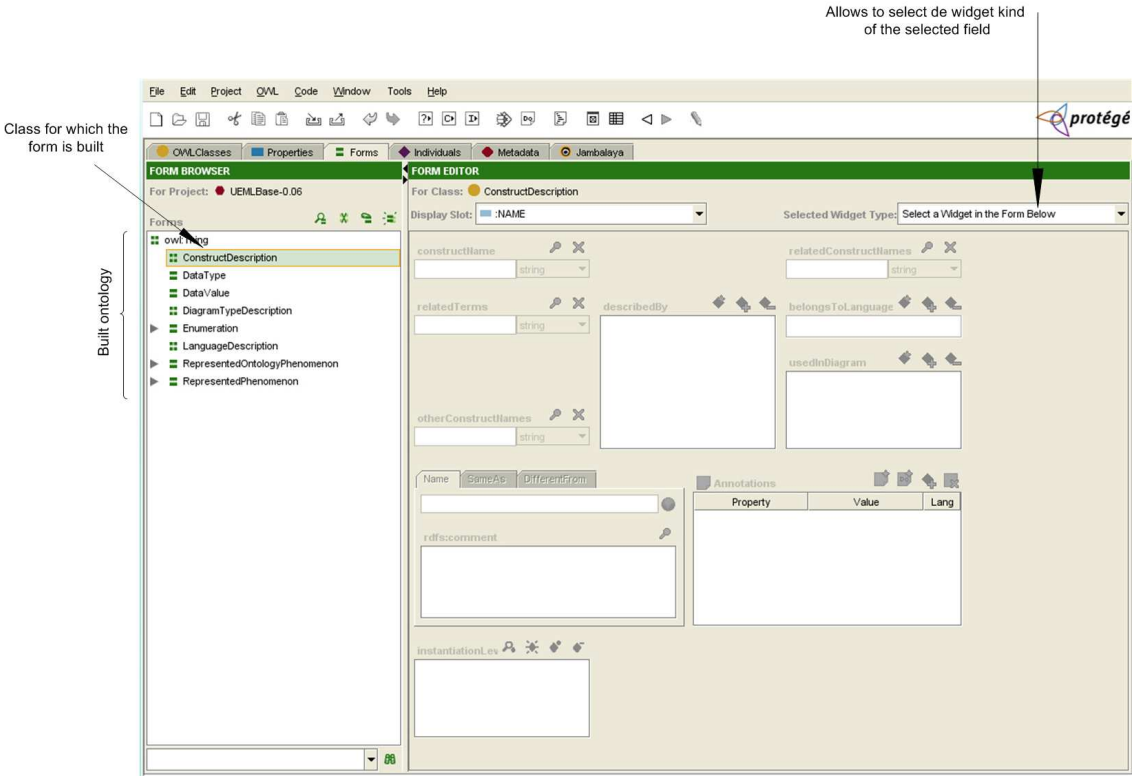


Figure 5.2: The Protégé “form” tab.

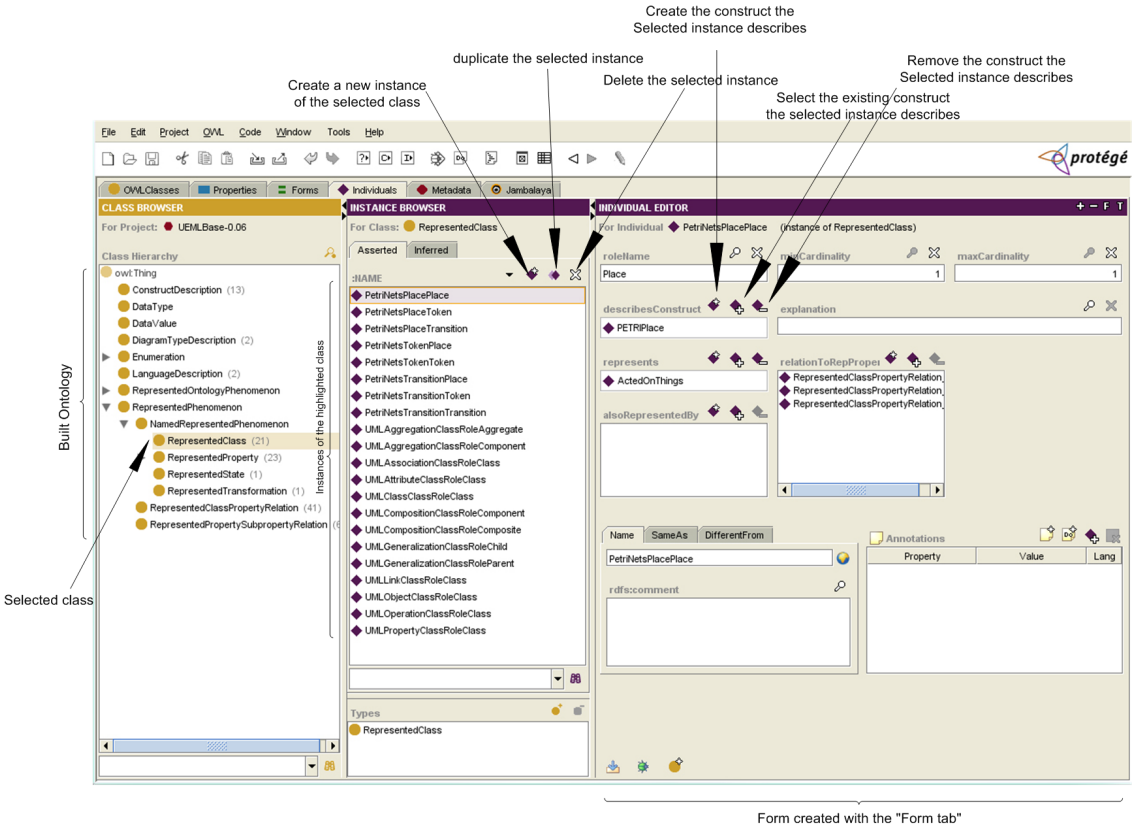


Figure 5.3: The Protégé “individuals” tab.

Chapter 6

The UEML 2.0 Meta-Meta-Model

In Chapter 4 we presented the UEML 2.0 template that briefly introduced what we call the UEML meta-meta-model. In this chapter we are going to explore in more detail this meta-meta-model written in OWL in drawn as a UML class diagram. To do this, we need to introduce the BWW model on which the ontology part of the meta-meta-model is based. After we will explain the role of each class and the basic constraints that are added to the meta-meta-model.

6.1 BWW Model

The BWW model of IS was proposed by Wand and Weber [WW93, WW95]. It is based on Mario Bunge's comprehensive philosophical ontology [Bun77, Bun79]. It is an already populated ontology which can be used to describe and analyze modelling constructs. Bunge's comprehensive philosophical ontology is a typical philosophical ontology (see Chapter 5). The BWW model is still a philosphical ontology even if it restricts the Bunge's ontology to the domain of IS. As we will see, in the UEML 2.0 meta-meta-model, the BWW model is formalized and extended. We can characterize this version as a computer science ontology.

6.1.1 Why the BWW Model?

[OHS04] explains that the BWW model has already been used to analyse and evaluate the modelling constructs of many established IS and EMLs, including:

- dataflow diagrams,
- ER models,
- NIAM,
- nine languages supported by the Upper CASE-toolset Excelerator,
- four languages supported the ARIS toolset for business modelling,
- the OPEN Modelling Language (OML),
- the Unified Modelling Language (UML)

The BWW model has also been used for general analyses of: IS design theory, object-oriented modelling constructs, systems decomposition, object-oriented IS, dimensions of data quality, optional properties in conceptual modelling, a two-layered information modelling approach where

instances are not tied to particular classes, whole-part relationships (like UML's aggregation and composition constructs) in OO models.

The BWW model is therefore a natural starting point for a template for defining enterprise modelling constructs, although alternatives exist both in the form of general philosophical ontologies, (e.g., [Chi96]), or special enterprise and IS ontologies, (e.g., the enterprise ontology [UKM98]) and the framework of IS concepts (FRISCO) [VS01]. In support of the BWW model, [WW93] have argued that Bunge's ontology is:

1. better developed and formalised than alternative philosophical ontologies,
2. based on concepts that are fundamental to the computer science and IS domains,
3. productive, in the sense that it has given useful results.

6.1.2 The BWW model concepts

In this section we present the most important BWW individuals that are used for the UEML 2.0 ontology. This presentation is largely inspired by [OHS05]. Table 6.1 gives definitions of all the BWW concepts used here.

Table 6.1: Basic concepts in the BWW model (from [OHS05]).

BWW concept	Concept definition
BWW-thing	"The elementary unit in our ontological model. The real world is made up of things." [WW95]
BWW-property of a thing	"Things possess properties" [WW95]. "We know about things in the world via their properties" [Web97].
Property precedence	"One properties precedes another if al the things that possess the latter property also possess the former." [Bun77].
BWW-complex property	"A complex BWW-property consists of other properties, which may themselves be complex" [Bun77].
BWW-property co-domain	"The set of values into which the function that stands for the property of a thing maps the thing" [WZ96].
BWW-class of things	"A set of things that can be defined by their possessing a particular set of properties" [WZ96]. 1) A BWW-class is defined by a "characteristic set" of properties. 2) All groups of BWW-properties that are possessed by at least one BWW-thing define a BWW-class.
BWW-subclass of things	"A set of things that can be defined via their possessing the set of properties in a class plus an additional set of properties" [WZ96]. (Hence, a BWW-subclass is itself a BWW-class.)
BWW-intrinsic property of a thing	"A property that is inherently a property of an individual thing" [WW95].
BWW-mutual property of two or more things	"A property that is meaningful only in the context of two or more things" [WW95].
BWW-state of a thing	"The vector of values for all property functions of a thing" [WW95].

BWW concept	Concept definition
BWW-state law of a thing	A property that "[r]estricts the values of the property functions of a thing to a subset that is deemed lawful because of natural laws or human laws" [WW95].
BWW-event in a thing	"A change of state of a thing. It is effected via a transformation" (see below) [WW95].
BWW-process in a thing	"An intrinsically ordered sequence of events on, or states of, a thing" [Gre96]. Processes may be either chains or trees of events [Bun77].
BWW-transformation of a thing	"A mapping from a domain comprising states to a codomain comprising states" [WW95].
BWW-transformation law of a thing	"Events are governed by transformation laws that define the allowed changes of state" [PW97]. [WW95] and other papers on the BWW model instead introduce BWW-lawful transformations, which define "which events in a thing that are lawful". The term "transformation law" instead of "lawful transformation" is chosen here to emphasise that a transformation law like a state law is a property of a particular thing.
BWW-law property of a thing	"Properties can be restricted by laws relating to one or several properties" [PW97]. 1) A law is either a state law or a transformation law of a particular thing. 2) A law is either a natural law or a human law (see below.)
BWW-composite thing	"A composite thing may be made up of other things (composite or primitive)" [WW95]. "Things can be combined to form a composite thing" [PW97].
BWW-component thing	Any BWW-thing that is in the composition of a composite thing.
BWWwhole-part relation	The property of being in the composition of another thing or, complementary, of having another thing as a component (according to [Bun77]).
BWW-resultant property of a composite thing	"A property of a composite thing that belongs to a component thing [WW95].
BWW-emergent property of a composite thing	A property of a composite thing that does not belong to a component thing (adapted from [WW95].)
BWW-history of a thing	"The chronologically ordered states that a thing traverses in time" [WZ96].
BWW-acting on another thing, BWW-coupling of things	"A thing acts on another thing if its existence affects the history of the other thing. The two things are said to be coupled [...]" [WW95].
BWW-direct acting on, BWW-binding mutual property	A thing acts directly on one or more other things when the former thing changes a BWW-binding mutual property they all possess. Changing the binding mutual property is an internal event in the former thing and an external event in each of the latter things.

According to Bunges ontology and the BWW model, there is a world that exists independently of human observers, and it consists of **things** that possess **properties**. *Atoms, fields, persons, artifacts and social systems* are examples of BWW-things. Conversely *properties of things* (e.g., energy), *changes in them*, or *ideas considered in themselves* are non-things. In particular, concepts are not BWW-things.

The BWW model distinguishes **properties** in different ways. A property can be **intrinsic** or **mutual**. A property is intrinsic if it belongs to only a single thing, whereas a mutual property belongs to two or more things. Intrinsic and mutual properties are perceived by humans in terms of attributes which are represented as functions over time. A **wholepart** relation is a property that relates an aggregate thing to one of its component things. A property can also be **resultant** if it belongs to a BWW-aggregate and is derived from one or more properties of its components or **emergent** if it belongs to a BWW-aggregate but not to any of its components. A **law** property restricts other properties of the same thing. A BWW-law is either a state law or a transition law. A property can be **individual** (or property of a particular) if it is a specific property like "being 25 years old" or "having grey hair", or **general** like "having an age" or "having a hair color." [Bun77] also distinguishes between BWW-properties that are **permanent** and those that are **variable**. BWW-properties can also be complex (composed of other properties).
A BWW-property precedes a second BWW-property if and only if:

- either the second property is complex and the first property is one of its constituents (*"being mammal" precedes "being human"*);
- or a BWW-law states that all BWW-things that possess the second property must also possess the first (*"being a human being" precedes "being married"*).

Things with a property in common form **classes**. A BWW-class is defined by a non-empty set of characteristic properties of the things in the class. The most general BWW-class is the class of all things, which is defined by the universal property of being able to associate with other things. Because characteristic properties may be complex, it is sometimes possible to say that a BWW-class is defined by a group of characteristic BWW-properties. One BWW-class may be defined by a group of characteristic properties that is contained in a larger group of properties that defines a second class. We then say that the second BWW-class is a **subclass** of the first.

A BWW-thing has time-dependent **states** that are determined by the values of the things property functions over time. A state can be **stable** (can only change as a result of an external action to a thing) or **unstable** (must change by a law). A change of BWW-state in a thing is an **event**, hence a BWW-event can be described as a pair of BWW-states. Consecutive BWW-events form complex events, or **processes** if they occur in the same thing. The sequence of consecutive BWW-states undergone by a thing (or, alternatively, the sequence of consecutive BWW-events) is called its history. A BWW-thing acts on a second thing if and only if the BWW-history of the second thing would have been different if the first thing had not existed. The first thing is called an **active** thing. Two BWW-things are **coupled** if and only if (at least) one of them acts on the other. BWW-couplings are caused by certain BWW-mutual properties that are said to be binding. A BWW-aggregate whose BWW-components are coupled is a system.

6.2 The meta-meta-model

In this section we present the meaning of each class of the meta-meta-model (Figures 6.1 and 6.2) and why they are interrelated. The meta-meta-model was entered in Protégé and thus written in OWL. It is also presented as a UML clas diagram for better readability. The meta-meta-model is divided into three main parts: the "preamble" part, the "represented" part and the "ontology" part. The "represented" and "ontology" parts correspond to the "representation" part of the

template and the "preamble" part to the "preamble" part of the template. The presentation part of the template is not taken into account in the meta-meta-model (see Section 3.2.2).

6.2.1 Preamble part

The first part ("preamble") is constituted of three classes: *LanguageDescrtiption*, *DiagramTypeDescription* and *ConstructDescription*. It is meant to contain general information about the language and the diagram type the modelling construct belongs to. A modelling construct belongs to one and only one language and can be used in several diagram types of the same language. We show here the description of the *DiagramTypeName*'s property in OWL:

```
<owl:DatatypeProperty rdf:ID="diagramTypeName">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
  <rdfs:domain rdf:resource="#DiagramTypeDescription"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

6.2.2 Represented part

The second part ("represented") is meant, as a structure, to contain a description of each modelling construct's semantics. A modelling construct is supposed to describe one or more phenomena. A phenomenon is "An appearance; anything visible; whatever, in matter or spirit, is apparent to, or is apprehended by, observation; as, the phenomena of heat, light, or electricity; phenomena of imagination or memory" [Wae03]. The represented part is useful to keep a link between what the construct represents in term of the common ontology and its intuitive signification. This level is thus an intermediate step between the construct and the ontology enabling one to understand the mapping. Indeed, as the ontology level is general it is not always easy to find the exact mapping. For instance, as we will see after, a GRL goal is notably mapped onto the *ActiveThings* class of the ontology. We understand why when we decompose the construct into the property goal characterizing an actor. This actor is actually an *ActiveThing* because attempting to reach a goal entails activity.

The phenomenon description is made with the help of the BWW model. Indeed, it is made in terms of *RepresentedClasses, RepresentedProperties, RepresentedStates* and *RepresentedTransformations* the construct describes. Each of them belongs to only one modelling construct. New *RepresentedClass, RepresentedProperty, RepresentedState* and *RepresentedTransformation* have thus to be invented each time a construct is described. This part is thus going to evolve often. The meaning of the classes is the following:

- A *RepresentedPhenomenon* is something represented by a construct. It can be either a *RepresentedClass*, a *RepresentedProperty*, a *RepresentedState* or a *RepresentedTransformation*. It has the following attributes:
  - explanation used to explain the role of the phenomenon;
  - minCardinality used to express the minimum number of instances of the phenomenon the construct represents;
  - maxCardinality used to express the maximum number of instance of the phenomenon the construct represents;
  - roleName used to name the phenomenon.
- A *RepresentedClass* is a class of things a construct represents (e.g., Actor or Token);

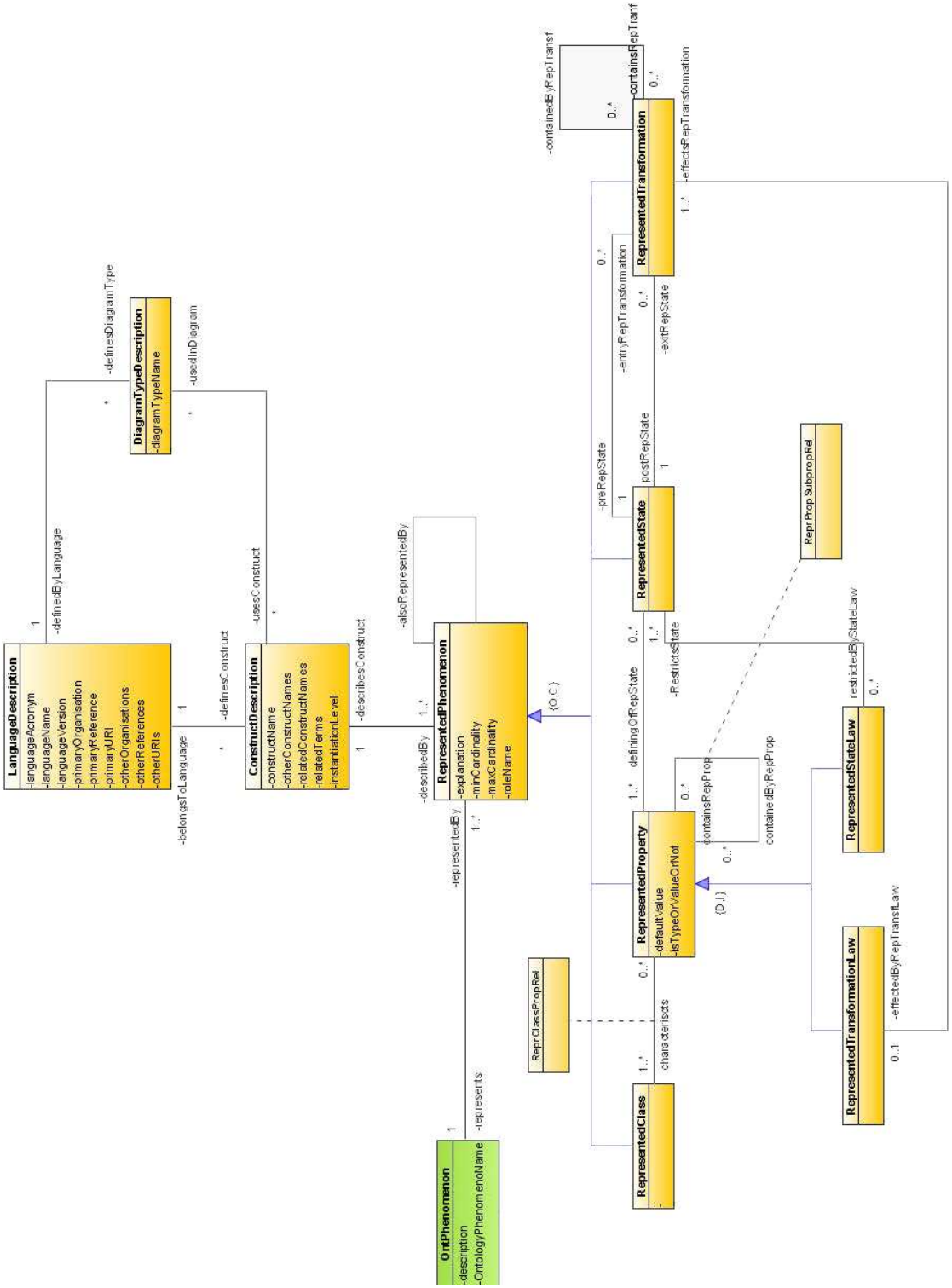


Figure 6.1: Upper part of the meta-meta-model.

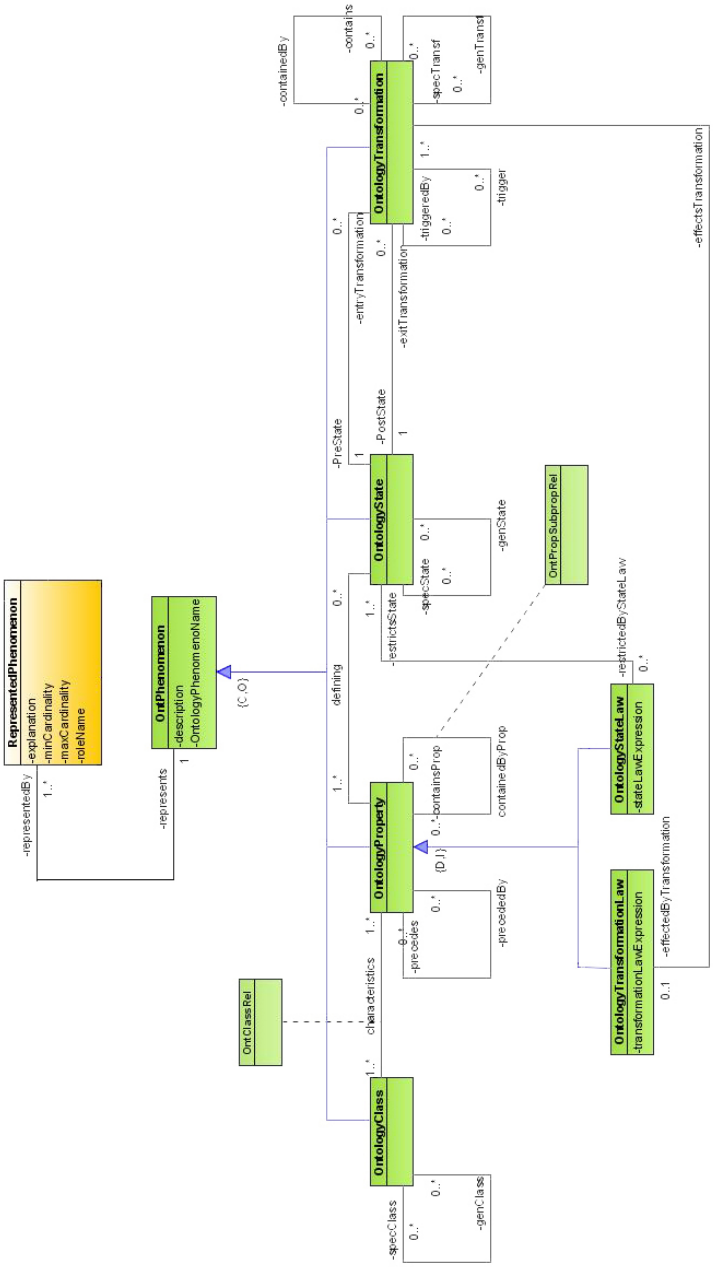


Figure 6.2: Lower part of the meta-meta-model.

- A *RepresentedProperty* is a property a construct represents (e.g., name, evaluation). The categorization distinguishes two special types of properties:
  - *TransformationLaw* that effects a *RepresentedTransformation*;
  - *StateLaw* that restricts a *RepresentedState*.
- A *RepresentedState* is a state represented by a construct (i.e., a vector of values for the properties the state is defined by).
- A *RepresentedTransformation* is a transformation represented by a construct (i.e., a mapping from a pre-state to a post-state).

As mentioned in Section 3.2.2 each modelling construct describes a Class because a property is the one of a class, a state is defined in term of properties that belong to a class and a transformation has pre- and post-states that are defined in term of properties that belong to a class.

6.2.3 Ontology part

The third part is the ontology. This is where the BWW classes are used to describe what the constructs represent in the real world. We build thus here an ontology which domain is what the EMLs altogether represent. The ontology is in essence common to every modelling construct of every language. It is not supposed to change as often as the "represented" part.

The ontology part is organized in the same way as the represented part. However, it is more precise. *Classes, States* and *Transformation* have a relationship of specialization/generalization between them (see Figure 6.2). Properties do not have it because specialization is defined in term of properties things have in common. Properties have a relationship of precedence between them (as explained in section 6.1). The specialization/generalization is not managed at the represented level for more facility. The most important is to have this relation at the ontology level because that is where modelling constructs can be compared. Indeed, every "represented" phenomenon has to represent an "ontology" phenomenon (of the same type). In the meta-meta-model (Figures 6.1 and 6.2) we only see one link (represents) between*RepresentsPhenomenon* and *OntologyPhenomenon* but it is actually more precise. A *RepresentedClass* can only represents an *OntologyClass*, a *RepresentedState* can only represent an *OntologyState*, etc... (see constraints in Section 8.3)

The ontology was first populated with classes, properties, states and transformations from Bunge's ontology and the BWW Model such as the classes *AllThings, AssociatedThings* and *AttributedThings* or the properties *AnyRegularProperty, AnyMutualProperty* and *AnyTransformationLaw*. As new constructs are mapped, the ontology can grow up as more specific classes, properties, states and transformations are included.

6.3 Example of a modelling construct description

In order to better understand how to describe a modelling construct in the meta-meta-model, we give the example of the "Goal" construct of GRL (from [MHO06]). For clarity, a standard way of graphical representation has been introduced. It enables to draw the represented classes, properties, states and transformation as well as the common ontology. The icons used for that purpose are showed in Table 6.2. Normal arrows represent subrelations and striped arrows mapping to the common ontology.

GRL goal is played by an "ontological scene" called "theGoal" and presented in Figure 6.3. The "ontological scene" consists of a set of represented classes and represented properties mapped to the common ontology. For `theGoal` which is a complex law property, the ontological scene consists of three parts: a thing `theGoal` belongs to, `theGoal` as a complex property, and `theGoal` as a law property.

Table 6.2: UEML 2.0 Graphical representation standard

	Instance	Type	Both	Common ontology
Things	Represented class name	Represented class name	Represented class name	Class name
Properties	Represented property name	Represented property name	Represented property name	Property name
States	Represented state name	Represented state name	Represented state name	State name
Transformations	Represented transform. name	Represented transform. name	Represented transform. name	Transformation name

Thing theGoal belongs to

GRL goal describes the intentions of some BWW thing that wishes the goal would become true. This thing is identified as the `actor` who holds the Goal. `Actor` is mapped to the `ActiveThings` because attempting to reach a goal entails activity.

theGoal as a complex property

Complex property is a BWW property that has subproperties. GRL goal is a complex property because it is characterised by different attributes that are all `anyRegularProperties` in the BWW model. Attributes for GRL goal include name, description, and evaluation.

theGoal as a law property

Law is a property "that restricts values that other properties can have". Goal expresses constraints on the possible states of a thing which might denote the proposed system, the entire organisation or a particular actor, and, thus, is mapped to `componentThings`, the least general BWW concept accounting for the three possibilities. Next `theGoal` is about this thing, meaning that attempting to reach a goal entails acting on this thing, which is also mapped to the `ActedOnThings` class in the common ontology. However in GRL there is no construct that indicates what this thing is. We describe it as the `thingGoalIsAbout`.

6.4 Constraints

In addition to the meta-meta-model itself, some constraints have been added in order to structure the ontology. These constraints show how the entries depend on each other. They also formalize the template and thus help for later tool development by making it more clear. [OHS05] divide them in four groups: "constraints on names", "constraints on ontClasses and RepresentedClasses", "constraints on ontProperties and RepresentedProperties" and "constraints on States and Transformations". We present them here in English but they will be presented in a more formal way (OCL) with other constraints we provide in Chapter 8.

6.4.1 Constraints on names

The constraints of this group ensure that some names are unique. These constraints are simple identifier but it is necessary to express them because that notion does not exist in UML.

1. Two distinct *ConstructDescriptions* cannot have the same "constructName".
2. Two *ontClasses* cannot have the same "className".

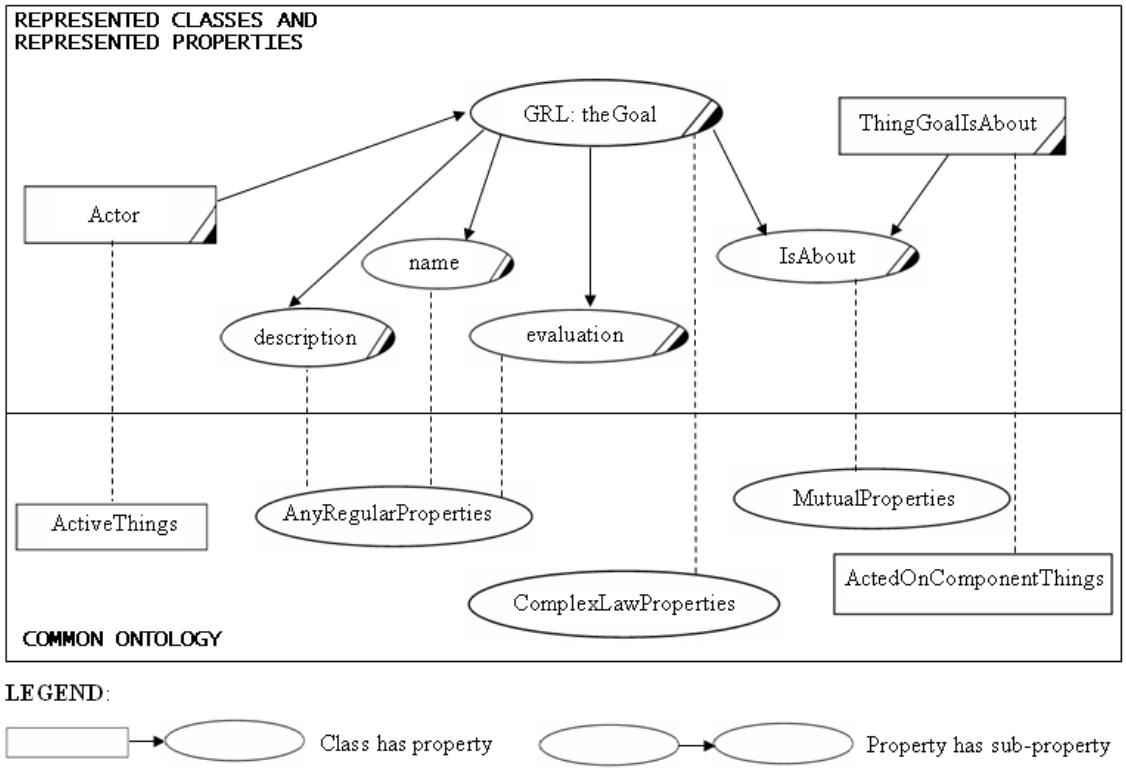


Figure 6.3: Simplified description of the GRL's Goal (from [MHO06]).

3. Two *ontProperties* cannot have the same "propertyName".
4. Two *States* with non-empty "Name" cannot have the same "Name".
5. Two *Transformations* with non-empty "Name" cannot have the same "Name".

6.4.2 Constraints on *ontClasses* and *RepresentedClasses*

The three constraints in this group deal with the uniqueness of "Class roleNames" within *ConstructDescription*, with the uniqueness of the set of "characteristic Properties" of an *ontClass* and with ontClass specialization/generalization.

6. If a *ConstructDescription* contains more than one RepresentedClass, each of them must have a roleName that is unique to the *ConstructDefinition*.
7. Two different *ontClasses* cannot be associated with the same sets of characteristic *ontProperties*.
8. If the set of "characteristic Properties" of one *ontClass* is a subset of that of another *ontClass*, the first *ontClass* must generalize the second.

6.4.3 Constraints on *ontProperties* and *RepresentedProperties*

The first four constraints in this group ensure that the *RepresentedClasses* and *RepresentedProperties* contained in a *ConstructDescription* match one another, i.e., that all the necessary *Classes*

and *Properties* are contained in the *ConstructDescription* and that the *Properties* belong to the *Classes* and vice versa. The last four ensure the uniqueness of *RepresentedProperties* "roleName" within a *RepresentedClass* and the consistency of the precedence relationship.

9. If a *ConstructDescription* contains a *RepresentedClass* that characterizes a *RepresentedProperty*, the *ConstructDescription* must also contain the *RepresentedProperty*.

10. Conversely, if a *ConstructDescription* contains a *RepresentedProperty* that characterizes a *RepresentedClass*, the *ConstructDescription* must also contain the *RepresentedClass*.
    `For instance, in GRL, a Goal is a complex property. This` ***`RepresentedProperty`***`, we call ‘‘theGoal" characterizes the` ***`RepresentedClass`*** `‘‘Actor". The construct has thus to be described by the` ***`RepresentedClass`*** `‘‘Actor" and the` ***`RepresentedProperty`*** `‘‘theGoal"`

11. If a *RepresentedClass* has a *RepresentedProperty*, the corresponding *ontClass* must have the corresponding *ontProperty* as "characteristic".
    `For instance, in GRL, the` ***`RepresentedProperty`***`, ‘‘isAbout" characterizes the` ***`RepresentedClass`*** `‘‘ThingsGoalIsAbout". As ‘‘ThingGoalIsAbout" is mapped onto the` ***`ontClass`*** `‘‘ActedOnComponentThings" and ‘‘isAbout" on the` ***`ontProperty`*** `‘‘MutualProperties", ‘‘MutualProperties" has to characterize ‘‘ActedOnComponentThings".`

12. Conversely, if a *RepresentedProperty* has a *RepresentedClass*, the corresponding *ontProperty* must be "characteristic" of the corresponding *ontClass*.

13. If a *RepresentedClass* has more than one *RepresentedProperty*, each of them must have a "roleName" that is unique relative to the *RepresentedClass*.

14. If an *ontClass* has a "characteristic Property" that is "preceded" by another *ontProperty*, then the *ontClass* must also have the second *ontProperty* as "characteristic".

15. If an *ontProperty* is "preceded" by a second *ontProperty* and the second *ontProperty* is "preceded" by a third, then the first *ontProperty* must also be "preceded" by the third *ontProperty* (transitivity of precedence).

16. An *ontProperty* cannot be "preceded" by itself (non reflexivity of precedence).

6.4.4 Constraints on *States* and *Transformations*

The final and most complex group of constraints deal with *States* and *ontTransformations*. The three first constraints deal with coherence between *State*, *Properties*, *Class* and *ConstructDescription*. The last one ensure‘s that every *ontTransformation* is different.

17. If an *ontState* has a set of *ontProperties*, there must be an *ontClass* whose set of "characteristic Properties" is a (possibly improper) superset of the first set.

18. If a *ConstructDescription* contains a *RepresentedState* and the corresponding *ontState* has a set of *ontProperties*, then there must be an *ontClass* whose set of characteristic Properties is a (possibly improper) superset of the first set and the *ConstructDescription* must contain the corresponding *RepresentedClass*.

19. If a *ConstructDescription* contains a *RepresentedState* and the corresponding *ontState* has a *ontProperty*, then the *ConstructDefinition* must also contain a corresponding *RepresentedProperty*.

20. Two distinct *Transformation* cannot have identical "from-" and "toStates".

6.5 Summary

In this chapter we presented the UEML meta-meta-model and the BWW-model on which it relies. We also provided an example of its use. The end of this chapter coincides with the end of the first part. In this first part, we learned what EMLs are and what are their problems caused by their diversity. We explored a solution to this problem with UEML and particularly its second version. UEML 2.0 relies on a template and a meta-meta-model that build an ontology based on the BWW model. This ontology constitutes a base of knowledge that enables the UEML applications (language comparison, model translation or language consistency checking). In the second part we will focus on tools that support UEML 2.0 and especially on the tool we made, a validator of the constraints.

Part II

Contribution

Chapter 7

Tools

In the previous part we learned what is UEML. Now, we are going to explore the tools that help achieve the objectives fixed by UEML 2.0. These tools assist one to follow the general approach we saw in Section 3.2.2 and to keep the data gathered from the analyses in a structured way.

7.1 Tools overview

In this section we present the tools that help to perform the analysis of languages constructs. Figure 7.1 depicts a typical scenario of applying UEML techniques and tools. This is an adaptation of the general method presented in Figure 3.3. The different steps are:

1. studying the language;
2. filling in the text-based template;
3. entering the data into Protégé ;
4. validating the construct defintions;
5. using the knowledge to compare, to translate or to check consistency.

The first step consisting in studying the language usually provides a meta-model of the language. This step is not usually supported by tools except for (meta-) modelling tools which do not offer particular guidance.

Secondly, one has to fill in the text-based template presented in chapter 4. This step is not yet assisted by a tool. For this step, a tool could help to select the ontology classes a construct represent, and to graphically do the analyses instead of writing everything.

Thirdly, in order to have a formalized knowledge, data have to be entered in the Protégé tool (that we will call Protégé UEML Tool from now on). Results are then put in a central base representing the meta-meta-model. This central base is automatically written in OWL by Protégé.

The fourth step consists in validating the analysis. Validation is performed using "UEML Validator", one of the major contribution of this work. This tool is described in Chapter 8.

Finally, a last tool called "UEML Semantic template manager" is under development. Its goal is to manage the data instead of the Protégé UEML Tool. It could also enable the update of the knowledge base from a distant location by several people at the same time.

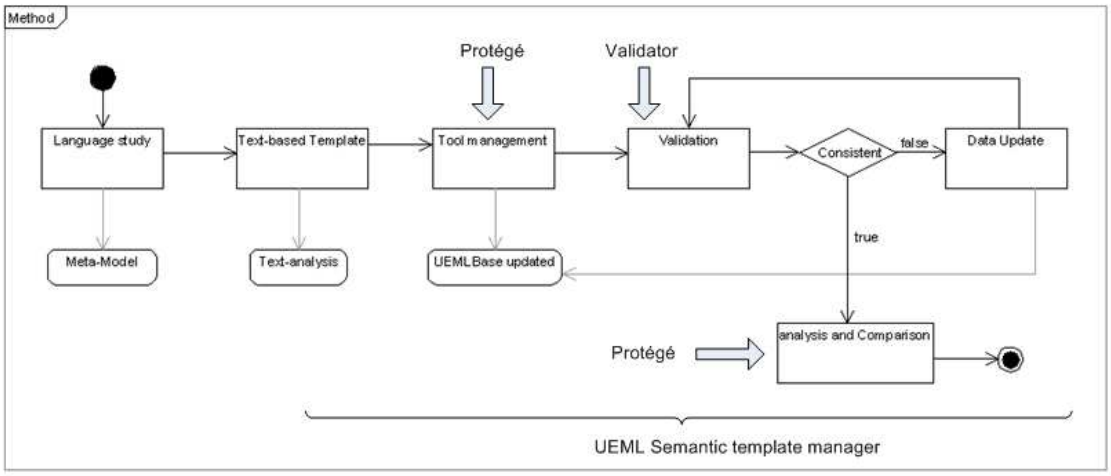


Figure 7.1: Typical scenario of applying UEML techniques and tools.

The main requirements of the three tools are:

Requirement 1 to gather the results of the construct analyses;

Requirement 2 to check the construct analyses;

Requirement 3 to enable an easy and distant update of the knowledge base;

Requirement 4 formalize the construct analyses;

Requirement 5 to enable an easy comparison.

7.2 UEMLBase

The Protégé UEML Tool is used for gathering all the analyses in one unique base and to formalize the results. Filling in the template with the Protégé tool is not really different than answering to the question of the text-based template, but with this new tool some improvements have been brought. First, the knowledge about the analyzed constructs are all in one OWL file managed by the tool. Then the analysis is more standardised as the data are managed by a tool. However, the Protégé version of the template only deals with the Preamble and Representation sections of the general approach (Section 3.2.2).

In order to formalize the analyses, the meta-meta-model has been described in the Protégé UEML Tool. Forms were also elaborated in order to collect the analyses data. With this description and these forms, Protégé allows one to enter the results of his analysis in a central OWL file.

As we see in Figure 7.2 the Protégé UEML Tool is composed of the different tabs we saw in Section 5.3.1. In order to enter the data, we thus have to use the "Individual" tab. In this tab one can choose the form of the kind of thing he wants to add an instance (e.g. *RepresentedClass* or *ConstructDescription*). In each form one has to fill in the different fields.

Protégé does not really help to do the analysis. The best way to use it is to first follow the template (e.g., by doing the text-based analysis) and then to add the results into the tool.

For the comparison, every analysed data is in the same base and can easily be consulted. However nothing is done to facilitate the finding of constructs respecting some constraints (e.g., every construct representing one *ontologyClass*).

Entering the data is easy but there is nothing to check if one forgot something mandatory or if something does not comply with the constraints we presented in Section 6.4. Indeed, even mandatory fields or relationships of an entity are not checked (this is partially due to the fact that we need a form for each different entity).

The most important contributions of the Protégé tool are thus to have a unique base of the whole knowledge (Requirement 1) and to formalize the analyses (Requirement 4). This knowledge base, coded in an OWL file, is called the UEMLBase.

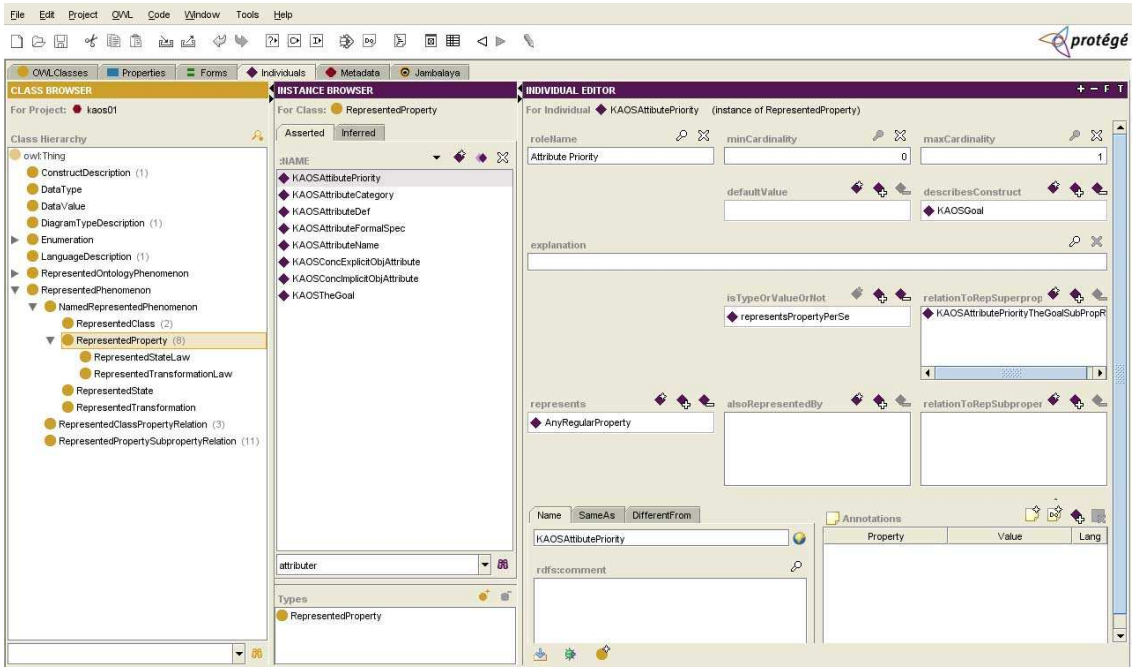


Figure 7.2: Protégé UEML Tool.

7.3 UEML Validator

The purpose of "UEML Validator" is to check the consistency of the analyses (i.e., if all the constraints presented in section 8.3 are respected). For each constraint it gives a list of the instantiations that do not respect it.

The main goal is to meet Requirement 2 and thus to get a consistent base at the end of the process. However there are also other objectives as emphazing "usual" mistakes, finding a list of mistakes that can be avoided by the new user interface or testing the "correctness" of the meta-meta-model. As "UEML Validator" is the work we have done, we present it much more deeply in chapter 8.

7.4 The "UEML Semantic template manager"

With the Protégé UEML Tool, construct analyses are managed in a central base that enables the use of the global knowledge. However, a problem remains: collaboration. Indeed, with the UEMLBase we can distribute the OWL file but it is impossible to update it simultaneously. As many people are involved in the process of languages analysis, the update of the central UEMLBase is problematic. The "UEML Semantic template manager " is a tool for publishing and exchanging

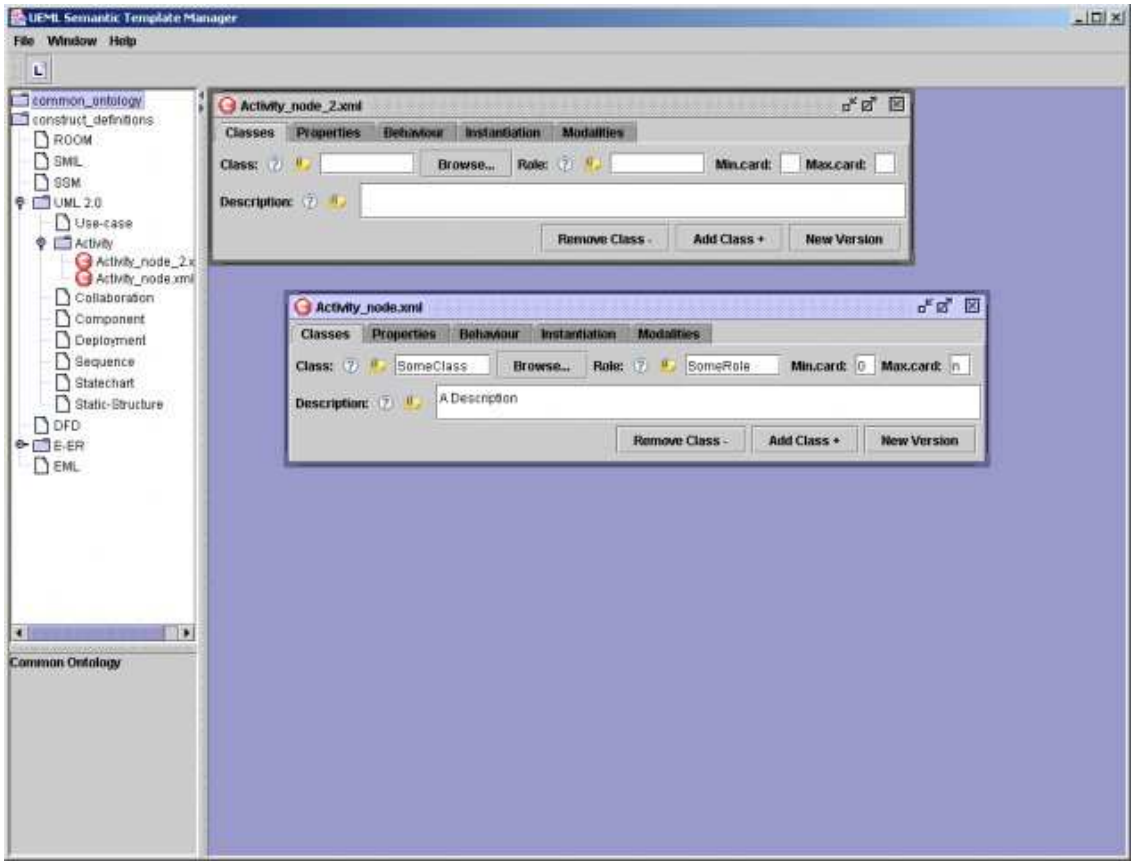


Figure 7.3: The UEML Semantic template Manager.

UEML templates. It is still a project under construction. It would replace the Protégé UEML Tool and provide other possibilities like the support of distant work. A prototype graphical user interface has already been developed (Figure 7.3) by Torbjørn Vefring a student of Pr. Opdahl. The solution is planned to be an online system for specification, sharing and editing of language constructs. Initially the system will be used by a few researchers but a goal is that it can be used by knowledgeable people in the role of language managers. The system to be developed should be a tool that supports asynchronous distant work and possibly features validation of input and version control.

This will require a design that is intuitive even for persons not familiar with the system; it should have an intuitive interface and actually aid the users in their primary goal by contributing to make the work less error prone, improve cooperation and thus make the prototype cost effective.

The main function of the system is to implement the meta-meta-model which allows language managers to create relationships across EMLs. “UEML Semantic template manager” should contribute to more effective information sharing through improving the information flow among users. It might also reduce repetitive or manual work, hence reducing the risk of errors and possibly improving the work environment for the end-user.

7.5 Summary

In this chapter, we saw the main tools supporting the UEML 2.0 approach. They help one to perform a language analysis compliant with the UEML approach and to manage the results of

these analyses. In the next chapter we will see the tool we built in order to face a problem of the Protégé UEML Tool: almost nothing is validated.

Chapter 8

The UEML Validator

In the previous chapter we presented an overview of the tools that help to support the UEML 2.0 approach. In this chapter we are going to investigate in details the tool called "UEML Validator". We first remind its objectives before explaining how it has been implemented. We will then see the constraints it checks.

8.1 Purpose

As we saw in the previous chapter, "UEML Validator" aims to check the UEMLBase consistency. It concentrates on Requirement 2 (see Section 7.1). For this, it checks if the constraints we present in section 8.3 are respected in the whole OWL file containing the analyses results. As already mentioned in the previous chapter, "UEML Validator" also has secondary objectives such as emphazing "usual" mistakes in order to make a list of errors that can be avoided by UEML Semantic template manager or testing the consistency of the model.

8.2 How it works

In this section we explain the architecture and the way the "UEML Validator" works. We also give instructions on how to write and comment new rules that could be checked by "UEML Validator". Indeed, as we will explain later, the tool is independent from the model and the rules. Rules can thus be removed or added easily depending on the model and its evolution.

8.2.1 Architecture

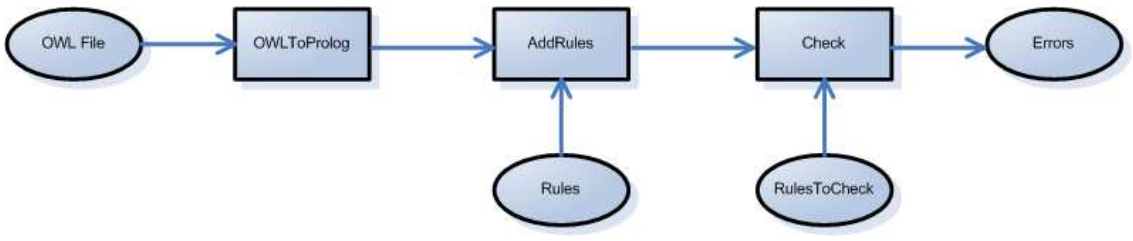


Figure 8.1: UEML Validator's pipes and filters architecture.

The "UEML Validator" has a pipe and filter architecture. Indeed, as we can see in Figure 8.1 it takes the informations of the OWL file in order to make a Prolog fact base. Then it adds the rules written in a separate file and finally, it checks them with the help of another (see section 8.2.4) file and shows the errors.

The "UEML Validator" has been made as independent from the model and the constraints as possible. The model of which we want to check constraints is not important. The only thing it has to respect is to have been written in OWL. Indeed, it generates the prolog base without knowing the name of the classes. The model has thus just to be written in OWL and to respect its syntax. This is the same for the rules that have to be checked. But even so, they have to respect two constraints:

1. to be written in Prolog;
2. to correspond to the way facts are generated (see section 8.2.3).

The model and the rules are not hard coded in the program so they have to be given to the program.

8.2.2 Implementation

"UEML Validator"'s implementation is rather simple. Indeed, the difficult part of the work was to find the constraints (see Sections 8.3 and 6.4.1) and to state them in Prolog. "UEML Validator" uses two main libraries: *Jena* [Lib] for the OWL extraction and *JPL* [Lp] for the Prolog part.

The "UEML Validator" class diagram is depicted in Figure 8.2. The main class of the program is `OC_ConstraintChecker.` It is the one that manages the graphical interface and call the different methods in order to check the model given in the graphical interface. The `OWLParser` class is the one using *Jena*. It is used to transform the OWL file of the model into a Prolog fact base. *Jena* enables one to get individuals from the OWL files in a logical way (without having to parse the whole file manually). The fact base is given to Prolog thanks to the `importFacts` method of the `Prolog` class. This `Prolog` class is actually the one that makes the link with the SWI-Prolog engine thanks to *JPL*. It enables one to send queries to the Prolog engine and get its answers. In order to check the constraints on the fact base, the `Checker` class takes each constraints from the `rules` and `rulesToCheck` files and gives them to the `Prolog` class. The Prolog answers are then given back to the main class that displays them.

8.2.3 Prolog base generation

As we wanted an independent and reusable tool we had to decide of a standard way to generate the Prolog facts from the instances of the OWL file. It is necessary because without knowing how facts are generated, it is impossible to write the rules. An OWL file is composed of three main concepts: classes, object properties and datatype properties.

The instances of classes produce this kind of Prolog fact:
nameoftheclass(nameoftheinstance).
The instances of object properties are written:
objectpropertyname(nameoftheinstance, valueoftheproperty).
Finally, as datatype properties are usually used to describe properties of a class, they are written:
nameoftheclass-datatypepropertyname(nameoftheinstance,valueoftheproperty).

The following simple OWL description can be represented with an object diagram (Figure 8.3).

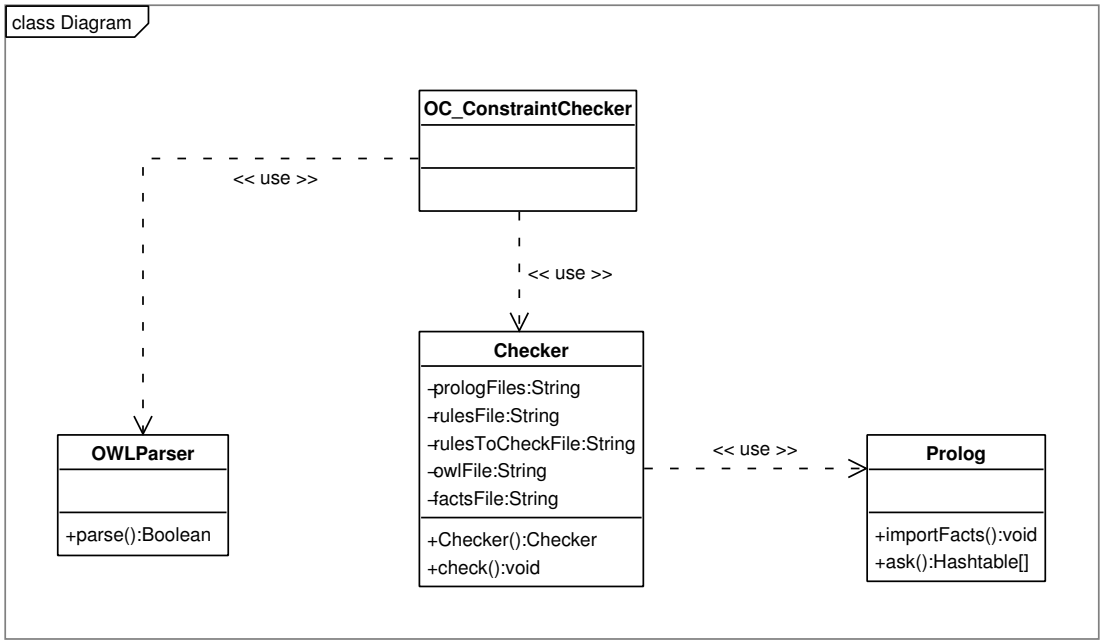


Figure 8.2: UEML Validator class diagram.

```
<DiagramTypeDescription rdf:ID="GRLModel">
  <usesConstruct rdf:resource="#GRLSoftGoal"/>
  <diagramTypeName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    GRL_Model
  </diagramTypeName>
</DiagramTypeDescription>

<ConstructDescription rdf:ID="GRLSoftGoal">
  <constructName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    SoftGoal
  </constructName>
  <usedInDiagram rdf:resource="#GRLModel"/>
</ConstructDescription>
```

This example would generate those Prolog facts:

```
diagramtypedescription(grlmodel).
constructdescription(grlsoftgoal).
usesconstruct(grlmodel,grlsoftgoal).
usedbydiagram(grlsoftgoal,grlmodel).
diagramtypedescription-diagramtypename(grlmodel,grlmodel).
constructdescription-constructname(grlsoftgoal,softgoal).
```

We provide an excerpt of the facts generated by UEML Validator with the UEMLBase 0.06 in Appendix C.

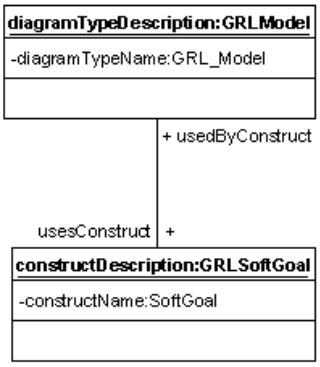


Figure 8.3: Example of Prolog facts generation from an OWL file as an Object Diagram.

8.2.4 The rules

The rules represent the constraints that "UEML Validator" has to check. They are are written in Prolog in a separate file. The only restrictions for the rules are to respect the Prolog's syntax and to correspond to the model. Intermediate rules that would help to write the rules one wants to check can be written in the same file. In order to check the selected rules (i.e., the constraints and not the intermediate rules) another file is needed. This file is called "rules to check". In this file, the rules to check are specified and described in a few words. Those few words are those that will be displayed by "UEML Validator" when giving the results. This file has to respect the following syntax:

```
''<DESCRIPTION>-<RULE><BR>''.
```

where <DESCRIPTION> is an explanation sentence that does not contain any dash and <RULE> is the name of the rule in Prolog. For instance, if we have those rules in the rules file:

```
hasDifferentProp(X,Y):- ontologyclass(X), ontologyclass(Y),
   not(X=Y), relationtoproperty(X,A), property(A,C),
   relationtoproperty(Y,B), property(B,D), not(C=D).

sameProperties(X,Y):- ontologyclass(X), relationtoproperty(X,_),
   ontologyclass(Y), relationtoproperty(Y,_), not(X=Y),
   not(hasDifferentProp(X,Y)).
```

We could have this following line in the "rules to check" file :

```
OntClasses having the same set of ontProp - sameProperties(X,Y).
```

This example is a possible "implementation" of the following rule: "Two different *ontClasses* cannot have the same sets of *ontProperties*" (rule 32 in Appendix A).

8.3 Implemented constraints

Before explaining the results we got with "Validator", we present the constraints it checks in the context of the meta-met-model. We defined 8 categories of constraints:

- "about mandatory fields", where the mandatory attributes for each entity are listed;

Table 8.1: Constraints group correspondences (the numbers in brackets represent the number of rule in each category).

New Constraints	Existing Constraints
Mandatory fields (8)	/
Relationships and cardinalities (19)	
Several ConstructDescription (1)	
Identifying names (4)	Names (5)
Classes (5)	ont- and Represented-Classes (3)
Properties (15)	ont- and Represented-Properties (8)
States (8)	States and Transformation (4)
Transformation (3)	

- "about relationships and cardinalities", that deal with constraints concerning direct relationships between entities;
- "about Identifying names", where names that have to be unique are given;
- "about *Classes*", where we present constraints related to *Represented* - and *ontClasses*;
- "about *Properties*", where we present constraints related to *Represented* - and *ontProperties*;
- "about *States*", where we present constraints related to *Represented* - and *ontStates*;
- "about *Transformations*", where we present constraints related to *Represented* - and *ontTransformations*;
- "about several *ConstructDescriptions*", that deals with constraints related to several *ConstructDescriptions*.

In Table 8.1 we show the correspondences between the constraints described in Section 6.4.1 and the ones presented here. All the rules presented in Section 6.4.1 are included in the next ones. The rules not mentioned before were written during our internship. The entire set of rules is described in Appendix A. We present here some rules from every category. Each rule is first described in English and then expressed in OCL [OMG05]. We express them in OCL because we used UML class diagram to represent the meta-meta-model and because OCL is the standard way to express constraints and queries.

8.3.1 Mandatory fields

In this section we give the list of mandatory attributes for each entity. "Validator" shows an error if one those fields is empty.

LanguageDescription

- languageName
- languageVersion

Context: LanguageDescription
inv: languageName.notEmpty() AND languageVersion.notEmpty()

8.3.2 Relationships between entities

In this section we list the constraints concerning direct relationships between entities. Those constraints are a translation of the cardinalities we can find on the meta-meta-model. They have to be checked by "Validator" because there is nothing to check it within OWL. As for the previous section, OCL expression is not given for every constraint for more clarity.

- Each *DiagramTypeDescription* has to be defined by a language.

Context: DiagramTypeDescription
inv: definedByLanguage.notEmpty()

- Each *ConstructDescription* has to belong to a language and has to be used by a *diagramTypeDescription* of this language.

Context: ConstructDescription
inv: belongsToLanguage.notEmpty() AND
UsedInDiagram.notEmpty() AND
belongsToLanguage =
UsesDescription.definedByLanguage

- Each *OntTransformationLaw* must effect an *ontTransformation*.

Context: OntTransformationLaw
inv: effectsTransformation.notEmpty()

8.3.3 Identifying Names

In this section we give the list of names that have to be unique in a certain scope.

- Two *ConstructDescriptions* cannot have the same *constructName* inside the same language.

Context: LanguageDescription
inv: definesConstruct− >isUnique(constructName)

- The *roleName* of each *RepresentedPhenomenon* of a *ConstructDescription* must be different.

Context: ConstructDescription
inv: describedBy− >isUnique(roleName)

8.3.4 Classes

In this section we produce the constraints concerning represented and ontological classes. They deal with uniqueness of *ontClasses* in terms of *ontProperties* set and with the "generalization" and "represents" relationship.

- Two different *OntClasses* cannot have the same sets of *OntologyProperties*.

Context: OntClass
inv: self $->$ *forAll*(c1, c2 | c1 <> c2 *implies*
c1.characterizedBy <> c2.characterizedBy)

- If the set of *OntProperty* of an *OntClass* is a subset of the set of *OntProperty* of another *OntClass*, then the first *OntClass* must generalize the second one.

Context: OntClass
inv: self $->$ *forAll* (c1,c2 | c1.characterizedBy.includesAll(c2.characterizedBy)
implies c1.genClass(c2))

8.3.5 Properties

It this section the constraints concerning Properties are listed. They deal with the *RepresentedClass* and *RepresentedProperties* a *constructDescription* have to possess, the coherence between *Properties* sets of *RepresentedClass* and *OntClass*, the coherence between *Classes* of *RepresentedProperties* and *OntProperties*, the "precedence" relationship, the coherence of the "contain" relationship, the *StateLawProperties*, the *TransformationLawProperties* and with the "represent" relationship.

- A *ConstructDescription* must describe the *RepresentedProperties* of the *RepresentedClass* it describes and vice-versa.

Context: ConstructDescription
Inv: self.describedBy− >select(c: RepresentedClass |
c.characteristics.describesConstruct.includes(self))
and
self.describedBy− >select(c: RepresentedProperty |
c.characteristics.describesConstruct.includes(self))

- If a *RepresentedClass* has a *RepresentedProperty* then the corresponding *OntClass* must have the corresponding *OntProperty*.

Context: RepresentedClass
Inv: represents.characteristics.includesAll(self.characteristics.represents)

8.3.6 States

The constraints concerning States are concentrated in this section. Those constraints deal with the *Class* of *State*, the *Properties* of a *State*, and with the "generalization" and "represent" relationships.

- A *RepresentedState* must be restricted by a *RepresentedStateLawProperty*.

Context: RepresentedState
Inv: restrictedByStateLaw.notEmpty()

- If a *RepresentedState* has a set of *RepresentedProperties*, there must be a *RepresentedClass* whose set of characteristic *RepresentedProperties* is a (possibly improper) superset of the first set.

Context: RepresentedState
Inv: self $->$ *forAll*(rs | RepresentedClass $->$ *exists*(c |
c.characteristics.includesAll(rs.defining)))

8.3.7 Transformations

The constraints concerning Transformations ensure that two *OntTransformation* are not the same in term of from- and to-*State* and the coherence of the "contains" and "represents" relationships.

- Two distinct *OntTransformations* cannot have identical from- and to-States.

Context: OntTransformation
Inv: self $->$ *forAll*(t1,t2 | t1.preState=t2.preState *and*
t1.toState=t2.toState *implies* t1=t2)

8.3.8 Several ConstructDescriptions

- A *representedPhenomenon* can only be described by one construct.

Context: RepresentedPhenomenon
Inv: self.describedBy.size() = 1

8.4 Summary

In this chapter, we saw the purpose of "UEML Validator", how it works and or it has been implemented. We also produced a list of constraints concerning the UEML 2.0 meta-meta-model. These constraints were formalized with OCL. With this chapter, we arrive at the end of the second part which presented an overview of the tools that support the UEML 2.0 approach. The next part provides applications that illustrate the use of UEML and its tools.

Part III

Application

Chapter 9

Tool testing

In this chapter we present the results we generated while using "UEML Validator" on the UEMLBase 0.06 containing the UML Class diagram analysis. It shows how helpful it is in the case of UEML. Indeed, many constraints are really easy to forget or are not even known. Moreover, as we saw in Chapter 7, data have to be entered with the Protégé interface which does not help to link information.

We do not give explanations on the Class diagram analysis as we will do so in the next chapter for GRL. This analysis was used to debug "UEML Validator" and to prove the usefulness of it. We first present the mistakes "UEML Validator" found on UEMLBase 0.06 and then present an interpretation of these errors.

9.1 Kind of mistakes

We found some twelve different types of mistakes in the UEMLBase 0.06 containing the analysis of UML Class diagram made by Prof. Opdahl (see Figure 9.1). These mistakes exist because a human has to enter the data concerning the analysis and humans are not perfect. In addition, the data of UEMLBase 0.06 were entered by the creator of the meta-meta-model and its Protégé counterpart. We guess we would find much more mistakes in analyses coming from other people that could have a lack of knowledge.

The rules enforced correspond the the rules 8, 10, 11, 14, 32, 37, 38, 39, 41 and 42 in Appendix A. These mistakes mainly concerned relationships (e.g., a construct that does not belong to any language – rule 10), bad name of "foreign key" (the "relatedConstruct" field in the *ConstructDescription – rule 11* class) but also the ontology management (e.g., a *RepresentedClass* characterized by *RepresentedProperty* which corresponding *OntClass* is not characterized by the corresponding *OntProperty* – Rule 38).

9.2 Interpretation

"UEML Validator" can help the user because he does not exactly know what is already in the ontology. For example, it can show to the user that an *ontClass* he added could be the specialization of another he did not even know it existed.

The errors helped us to make a list of recommendations for the development of UEML Semantic template manager (see Section 9.3) and showed up an improvement to do in the design of the meta-meta-model.

The improvement is about *relatedConstructName* property of the *constructDescription* class. As this field is supposed to contain the other similar constructs descriptions of the language, they have to exist in the model. The best way to express that would be to have a relationship to these other constructs and not a field we can fill in manually. ¡

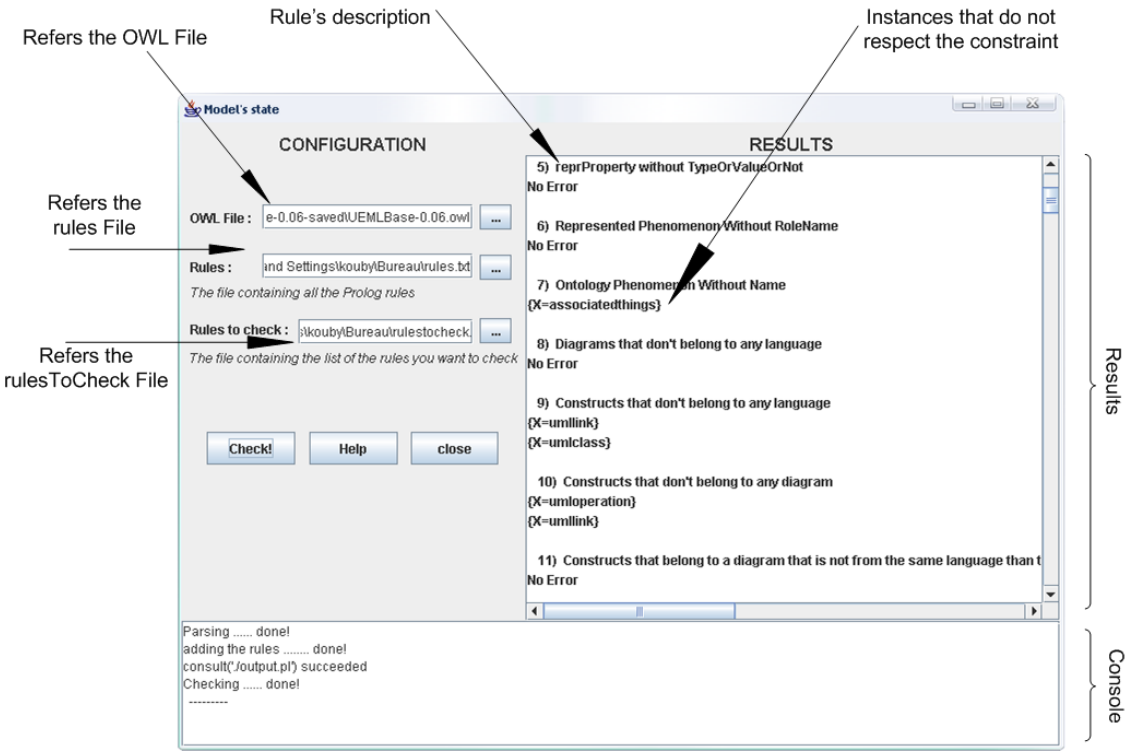


Figure 9.1: The UEML Validator.

9.3 Recommendations

In this section, we list what could be done with the UEML Semantic template manager in order to avoid the typical errors we found by using "UEML Validator" on UEMLBase 0.06.

9.3.1 Missing properties

The tool could easily warn the user when a mandatory field is not filled in. Here is a list of fields this tool should check (the numbers in brackets refer to the rules in Appendix A):

1. LanguageDescription : languageName, languageVersion (1,2);
2. DiagramTypeDescription : diagramTypeName (3);
3. ConstructDescription : constructName, instantiationLevel (4,5);
4. RepresentedPhenomenon : roleName, represents, TypeOrValueOrNot (6,7, 36, 49, 50, 51, 59, 61);
5. RepresentedProperty : relationToRepClass (14);

6. RepresentedTransformationLaw : effectsRepTransformation (19);
7. RepresentedStateLaw : restrictsRepState (18);
8. RepresentedTransformation : preRepState, postRepState (17);
9. OntologyPhenomenon : name (8);
10. OntologyProperty : relationToClass (22);
11. OntologyTransformationLaw : effectsTransformation (27);
12. OntologyStateLaw : restrictsState (25);
13. OntologyTransformation : preState, postState (23, 24).

9.3.2 Automatic additions

The tool could also propose to the user to add many things automatically. Indeed, depending on what the user has already entered, it can tell him to add some relationships or phenomena. Here is the list of these automatic additions:

1. When a *RepresentedClass* is said to describe a *ConstructDescription*, the GUI should add the *RepresentedProperties* of this *RepresentedClass* to the list of *RepresentedPhenomenon* that describe the *ConstructDescription* (37);
2. Conversely, when a *RepresentedProperty* is said to describe a *ConstructDescription*, the GUI should add the *RepresentedClass* of this *RepresentedProperty* (37);
3. When a *RepresentedState* is said to describe a *ConstructDescription*, the GUI should add the *RepresentedPoperties* with which this *RepresentedState* is defined to the list of *RepresentedPhenomenon* that describe the *ConstructDescription* (55);
4. When a *RepresentedProperty* is added to a *RepresentedClass*, the GUI should propose to add the corresponding *OntProperty* to the corresponding *OntClass* if it is not yet done (39);
5. When an *OntProperty* is said to characterize an *OntClass*, the GUI should propose to add the *OntProperties* that precede this *OntProperty* to the list of *OntProperty* that characterize this *OntClass* if it is not yet done (41);
6. When an *OntProperty* is said to be preceded by another *OntProperty*, the GUI should propose to add all the *OntProperty* that precede this other *OntProperty* to the list of *OntProperty* that precede this first *OntProperty* (42);
7. When a new *OntTransformation* is added, the GUI should check if the same does not already exist (having the same pre- and postConditions) (60).

In some cases, the UEML Semantic template manager should add automatically, and in other cases not. We think, it should not always add automatically because managing the ontology is not something easy. It needs a lot of reflexion. By proposing one to add something, the UEML Semantic template manager will maybe show him he is about to make a mistake instead confirming him in his opinion.

9.4 Summary

The use of "UEML Validator" on UEMLBase 0.06 showed the usefulness of the tool. Indeed, it found many mistakes while it was made by the creator of the UEML method and meta-meta-model. The results of "UEML Validator" also gave us the opportunity to make recommendations for the future UEML 2.0 tool. In the next chapter we will provide a complete analysis of the GRL Goal construct.

Chapter 10

Case Study

In this chapter, we present a UEML 2.0 application with its tools. The goal is to show how to use UEML and how important the tools are. We first explain the method we will follow, and then do the analysis. The language used for the purpose of the case study is GRL presented in Section 2.3.

10.1 The method

The purpose of the case study is to illustrate how tools, supporting the UEML approach, could be used for language analysis. [DHP05, MHO06] provide an analysis of the GRL constructs. In this section we will focus on the Goal construct. In order to perform the analysis of GRL Goal construct, we will follow the scenario depicted in section 7.1 that is a way of putting into practice the UEML strategy of Section 3.2.2. We will use the tools currently available (i.e., the Protégé UEML Tool and "UEML Validator"). We begin thus with the study of the language and the production of its meta-model. Then we follow the text-based template and enter the results into the Protégé UEMLBase. Finally we check the analysis consistency with "UEML Validator" and do the necessary improvements.

10.2 Language study

We already presented GRL in section 2.3. For this chapter we will only perform the analysis of the "Goal" construct. As explained in Section 6.3 GRL goal is played by an "ontological scene" called theGoal and presented in Figure 10.4. We also provide the GRL meta-model (Figure 10.2). This meta-model was made by [DHP05] who explains it as follow:

"Figure 10.1 gives the top-level view of the metamodel. The 4 main types of elements appear immediately: Actor, IntentionalElement, NonIntentionalElements, and link (that was renamed IntentionalRelationship, to be compliant with the GRL syntax definition). Instances of NonIntentionalElements are proxies used by GRL as a reference to constructs in different models. Figure 10.2 details the 5 kinds of intentional elements: SoftGoal, Resource, Task, Goal and Belief. it also introduces the various kinds of GRL relationship types as subclasses of IntentionalRelationship. How these interact with other classes is detailed in Figure 10.3. In Figure 10.2, various abstract classes appear: IEButBelief (denoting all intentional elements except beliefs), Correlator, Contributee, Contributor and Depender/Dependee. These latter classes are somewhat artificial. We introduced them for the sole purpose of showing graphically and succintly the groups of classes which are likely to play a role with respect to an intentional relationship. For example, the GRL

syntax requires that the contributee in a contribution relationship is either a belief, a softgoal or an intentional relationship. Therefore, we have introduced an abstract superclass Contributee generalizing Belief, SoftGoal and IntentionalRelationship. This way, modelling GRL relationship types becomes much easier. We can observe that in Figure 10.3. For example, we see that representing contribution relationships only requires one class (ContributionRelationship) and two associations (contributor and contributee) both pointing to abstract classes (respectively Contributor and Contributee). If we had not introduced these abstract classes, the associations would have been multiplied (one for each type of contributor and one for each type of contributee) and OCL constraints would have been necessary to exclude that a contribution relationship has more than one contributee. To distinguish them from the other more obvious classes, all such abstract superclasses have been given the stereotype "PossibleRole(s)" and were named after their corresponding roles (except IEButBelief, for brevity)."

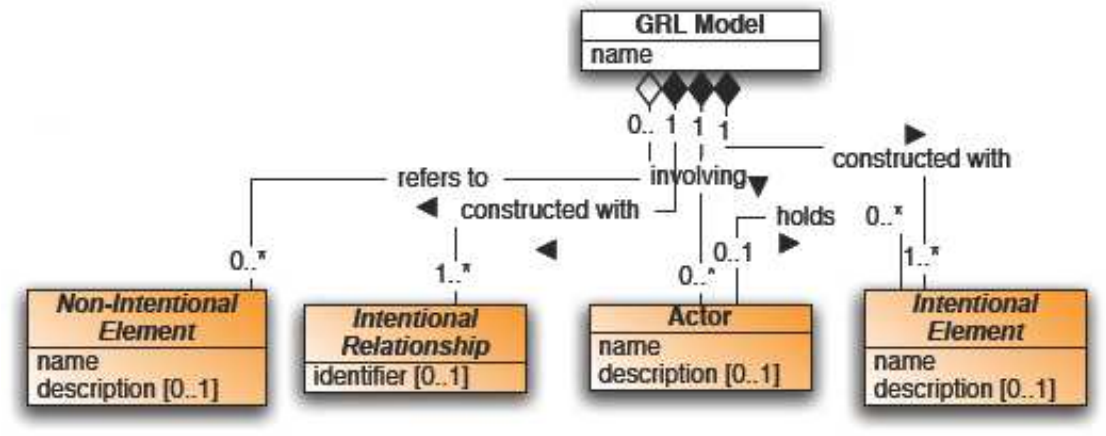


Figure 10.1: Top-Level view of the GRL Meta-model (from[DHP05]).

10.3 Text-based Template

In this section we provide the analysis done with the test-based template presented in Chapter 4. This is the results of the analysis done by [DHP05] and [MHO06].

10.3.1 Preamble

- **Construct name** – Goal
- **Alternative construct names** – Condition to achieve, State of affairs to achieve, Objective
- **Related, but distinct construct names** – SoftGoal
- **Related terms**
  - Intentional element: a goal is an intentional element. Intentional element is the set comprising SoftGoal, Ressource, Task, Goal and belief.
  - Sub-element: this is the role played by a goal which is decomposed in a decomposistion.
  - Dependum: that is the role played by a goal, a softgoal, a ressource or a task depended on in a dependency.

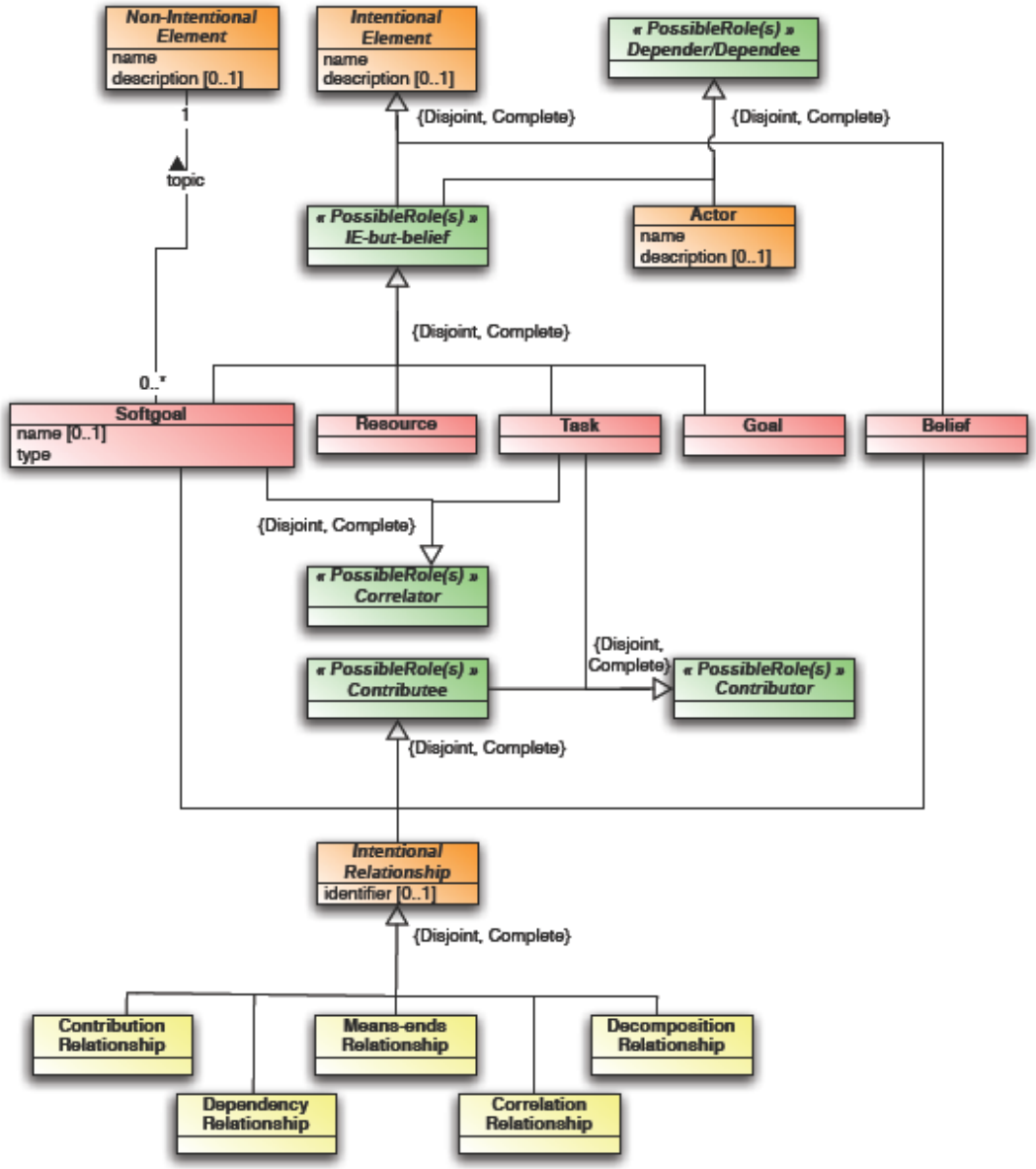


Figure 10.2: GRL Meta-model: zoom on intentional elements (from[DHP05]).

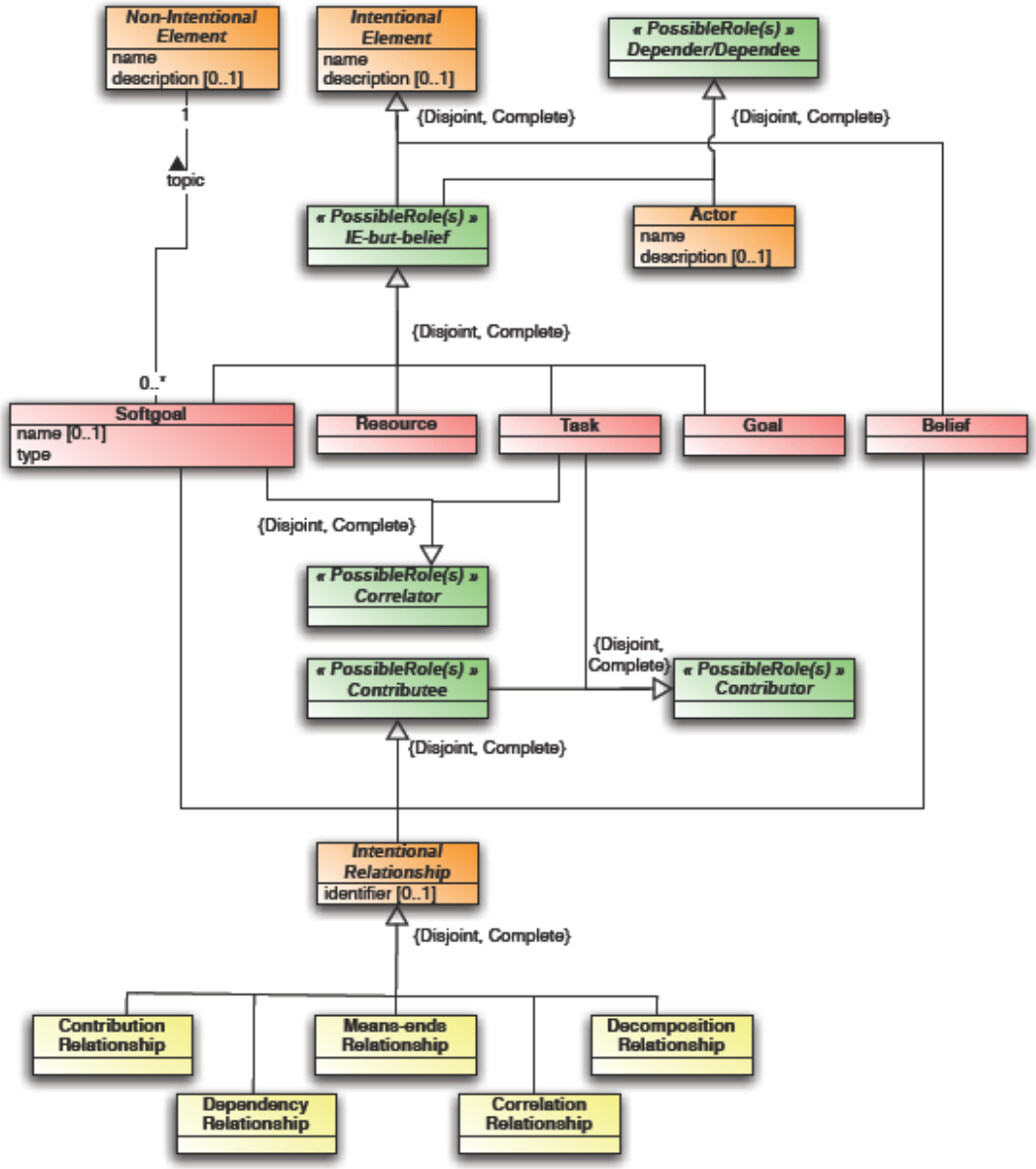


Figure 10.3: GRL Meta-model: zoom on intentional relationships (from[DHP05]).

  - End: this is the role played by a goal which is the objective achieved using task in a means-end link.

- **comments:** sometimes a goal plays the role of a depender or a dependee in a dependency relationship (if this goal is held by an actor). In this analysis we ignored this case.
- **Language** – Recommendation Z.151 (GRL) - Version 3.0, Sept. 2003
  http://www.usecasmaps.org/urn/z_151-ver3_0.zip
  Also called GRL or URN-NFR
- **Diagram types** – GRL Model (the only diagram type in GRL)

10.3.2 Presentation

- **Icon** – A goal is represented by an oval with the name inside and attributes between square brackets. Here is an example:

- **Builds on** – None
- **Built on by**
  - A dependency can have a goal dependum
  - A decomposition can have a goal as a decomposed element
  - A means-end can have a goal as end element
- **comments:** sometimes a goal plays the role of a depender or a dependee in a dependency relationship (if this goal is held by an actor). In this analysis we ignored this case.
- **User-definable attributes**
  - Name: the name of the goal
  - Description: an optional textual description of the goal
  - Evaluation:
  - Any other attribute the user wishes to add
- **Relationships to other constructs**
  - Belongs to 1..1 GRL Model
  - Can have 0..n Attribute
  - Can be held by 0..1 Actor
  - Can play the role of
    - a dependum in 0..n dependency links
    - a sub-element in 0..n decomposition link
    - an end element in 0..n means-ends link
- **Layout conventions** – Nothing particular

10.3.3 Representation

- **Instantiation level** – Both type and instance level.
- **Classes of things**
  - `ActiveThings` representing the actor holding the goal.
    - Cardinality 1-1
    - Role name: "Actor".
  - `ActedOnComponentThings` representing what the goal is about.
    - Cardinality 1-1
    - Role name: "ThingGoalIsAbout"
- **Properties (and relationships)**
  - `ComplexLawProperties` representing the goal
    - Cardinality: 1-1
    - Role name: "theGoal"
    - characterizes "Actor"
  - `AnyRegularProperties` representing the name of the goal
    - Cardinality: 1-1
    - Role name: "name"
    - characterizes "Actor"
    - subProperty of "theGoal"
  - `AnyRegularProperties` representing the evaluation of the goal
    - Cardinality: 1-1
    - Role name: "evaluation"
    - characterizes "Actor"
    - subProperty of "theGoal"
  - `AnyRegularProperties` representing the description of the goal
    - Cardinality: 1-1
    - Role name: "description"
    - characterizes "Actor"
    - subProperty of "theGoal"
  - `MutualProperties` representing the properties the goal is about
    - Cardinality: 1-1
    - Role name: "IsAbout"
    - characterizes "ThingGoalIsAbout", "Actor"
    - subproperty of "theGoal"
- **Behaviour** – Lifetime
- **Modality** – The holding actor wishes the state law represented by the goal to become true.

The Figure 10.4 depicts the phenomena described by the Goal construct as explained in the template.

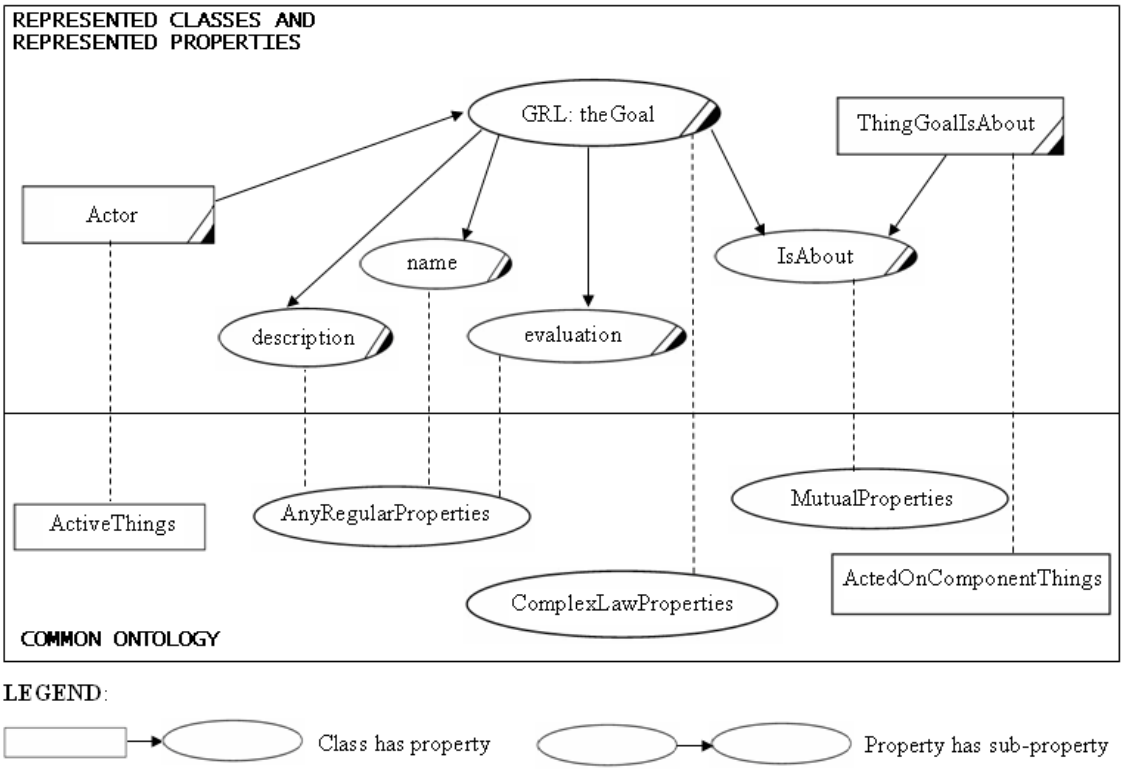


Figure 10.4: The phenomena represented by the "Goal" construct (from [MHO06]).

10.4 Protégé UEML Tool

Using the UEML Base consists in "translating" the text-based analysis. Indeed, the results of the previous step concerning the preamble and the representation parts are entered via the Protégé forms made for UEML 2.0. As explained in Section 7.2 the main goal is to have a common base for every construct of every language that allows to compare them. This step is not easy because of all the links existing between entities. We show in Figure 10.5 the data concerning the *RepresentedProperty* "theGoal".

10.5 UEML Validator

With "UEML Validator" we know if the data entered in the UEMLBase respect the constraints listed in Section 8.2.4. As we can see in Figure 10.6 (error 19), "UEML Validator" found a mistake on the analysis entered. It says that the *RepresentedProperties* `GrlGoalName, GrlGoalDescription` and `GrlGoalEvaluation` do not characterize any *RepresentedClass*. Indeed, every *Represented-Property* has to characterize a *RepresentedClass*. It is important because, then the corresponding *OntProperty* has to characterize to corresponding *OntClass*. It is important to know which kind of things the properties that describe a construct, characterize. In order to correct these mistakes we just have to add the link between the *RepresentedClass* `Actor` and the *RepresentedProperties* `GrlGoalName, GrlGoalDescription` and `GrlGoalEvaluation` . This relationship is not shown in Figure 10.4 for more clarity (the sub-properties characterize the same class than their sup-property, if nothing is said) but was already mentioned in the text-based analysis.

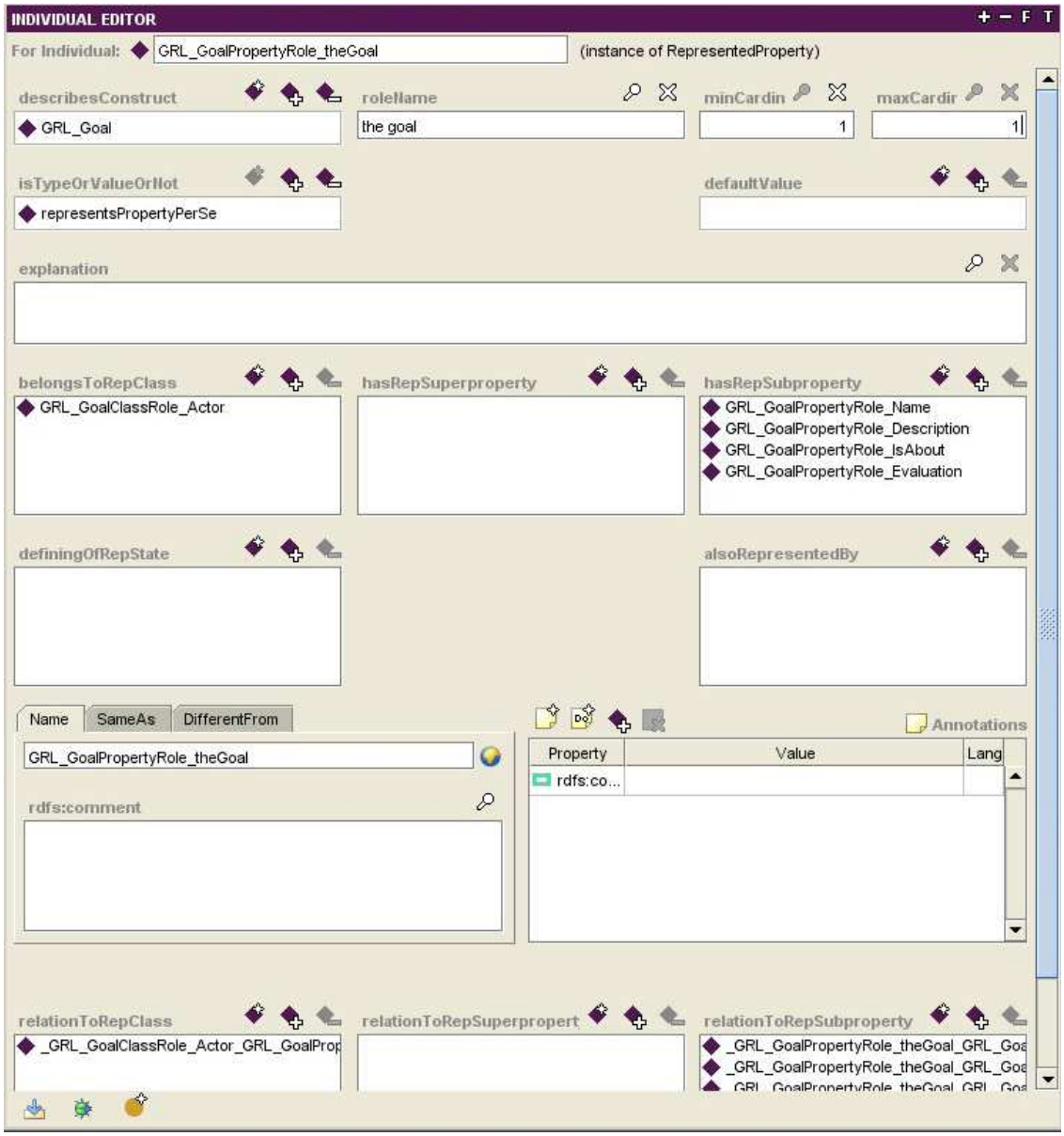


Figure 10.5: The "Goal" construct in the Protégé UEML Tool.

When entering the complete GRL analysis in the UEML Base, many mistakes were found by "UEML Validator". Theses are the mistakes number 7, 13, 18, 17, 20, 37, 38, 53, 54, 55 and 57 of Appendix A.

10.6 Observations

In this Section we will provide comments about the analysis and about "UEML Validator" behaviour. For the analysis we will discuss the mapping of `theGoal` and for "UEML Validator" we will discuss the results it does not provide.

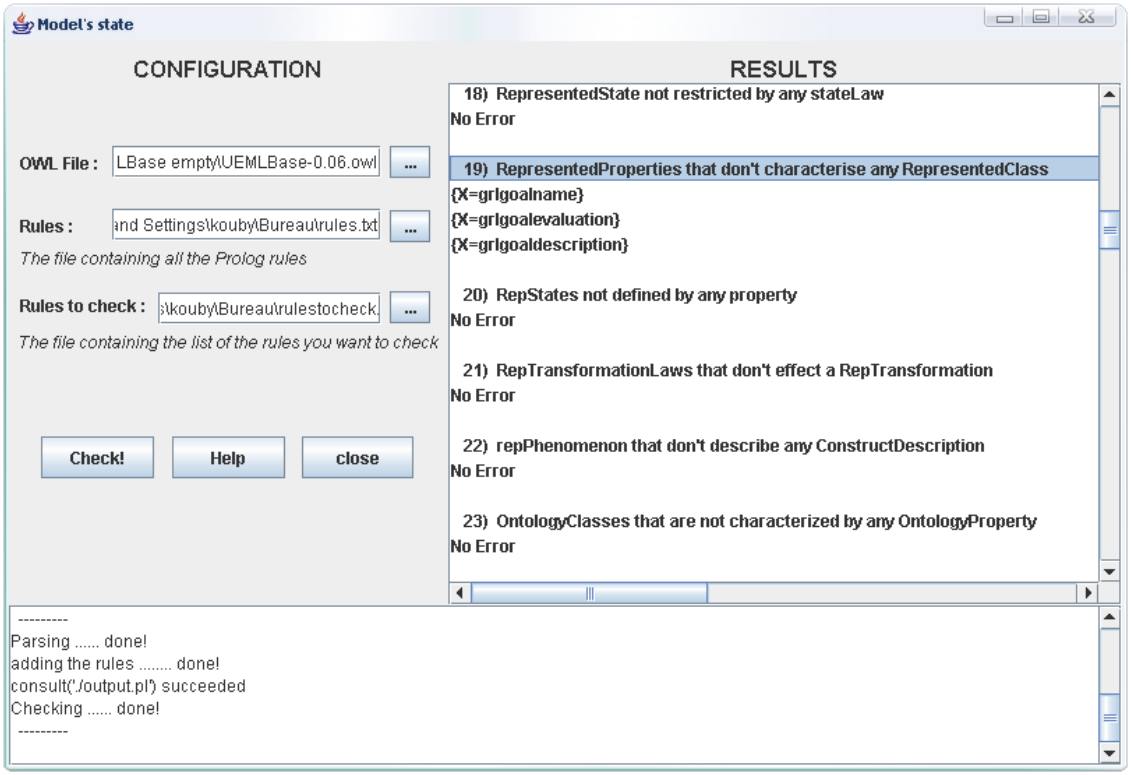


Figure 10.6: The error showed by "UEML Validator".

10.6.1 UEML Validator is not context-dependant

In the previous Section "UEML Validator" mentionned that the *RepresentedProperties* `description`, `name` and `evaluation` had to characterize a *RepresentedClass*. But it did not mention any error about the *RepresentedProperty* `IsAbout`. In truth, `Isabout` should also characterize the *RepresentedClass* `Actor` because it is a `MutualProperty`. "UEML Validator" is not context dependent. It knows the generic rule that each *RepresentedProperty* has to characterize a *RepresentedClass* but does not take into account on which instance of the ontology the *RepresentedPhenomenon* is mapped. This is why this error was not spoted by "UEML Validator".

10.6.2 theGoal as a *LawProperty*

In Section 10.3, we mapped `theGoal` on the *OntClass* `ComplexLawProperty`. However, a `LawProperty` is either a `stateLawProperty` or a `transformationLawProperty`. But in the case of GRL, a Goal is sometimes a `stateLawProperty` and sometimes a `transformationLawProperty`. Because of this, we cannot represent the state or transformation the construct may represent. This lead to a loss of information. [MHO06] depicts this situation and shows that KAOS [Lam03, Let01] divides these two cases by defining goal patterns: achieve and cease for *transformationLaws* and avoid and maintain for *stateLaws*. It and enables us not to loose any information in the mapping.

10.7 Summary

In this chapter we described how the UEML 2.0 approach works in practical terms. We also explained the use of the different tools. The application allowed us to find a mistake in our Protégé Base updated with the "Goal" analysis. This chapter closes the last part of the thesis. General conclusions are given in the next chapter.

Chapter 11

Conclusion

11.1 The problem

We noticed that enterprise modelling (EM) can be used by companies in every sector. It can also be used to model many different enterprise areas such as organisation, resources, information, requirements, goals and strategy. This extremely large scope gave rise to many different enterprise modelling languages (EML). The different enterprise models inside one enterprise are interrelated and enterprises cooperate more and more exchanging many of their models. But most EMLs are implemented in tools with proprietary terminology and modelling constructs. The different enterprise models are thus incompatible. A real need for EML interoperation appeared.

11.2 Contribution

The thesis investigated the solutions given by the two versions of Unified Enterprise Modelling Language (UEML) in the domain of EMLs interoperability. We briefly presented the first version before going into details in the second version. We explored the UEML 2.0 approach and more particularly its template and meta-meta-model.

The template is the method to follow in order to perform UEML 2.0 compliant EML analyses. In order to incorporate a language into UEML, each of its modelling constructs has to be described in a standard way defined by the template. It consists in a text-based form divided in 3 main parts: Preamble, Presentation and Representation. The preamble section deals with general issues about the modelling construct; the presentation section describes the visual presentation of the modelling construct and the representation section deals with the semantic aspects of the constructs.

The meta-meta-model enables one to keep the knowledge acquired thanks to the template and to build an ontology of what the EMLs can represent. This ontology relies on the BWW model but grows up specifically when adding new constructs descriptions.

The method aims to enable one to work with the knowledge acquired thanks to the template in order to build the UEML as a federator. But doing that manually becomes rapidly impossible. This is the reason why tools are necessary. We presented the existing tools and the tool we made ("UEML Validator"). The Protégé UEML Tool is very useful because it manages the whole ontology in one unique OWL file. "UEML Validator" enables one to check if the analysis done respects the constraints. These constraints restrict how the meta-meta-model can be populated and ensure it is more consistent.

Our work, and more particularly "UEML Validator", gave different kinds of results. First, we produced and formalized some sixty-five constraints that the instances of the meta-meta-

model have to respect. Secondly, the use of "UEML Validator" on UEMLBase 0.06 gave us the possibility to make recommendation for the tool in construction (UEML Semantic template manager). Finally, it enabled us to be sure the analysis of GRL is consistent and to demonstrate the utility of our validator.

11.3 Future works

Today, the main limitation for the UEML approach is the way the meta-meta-model can be populated. For the moment, the unique UEMLBase cannot be accessed from a distant location. Each analysis has to be entered in the Protégé UEML Tool by the same person. It is a brake to the UEMLBase development which need to be extended if we want to have interesting results. UEML Semantic template manager should solve the problem but a general agreement is still needed on who can do it and which languages have to be studied.

"UEML Validator"'s main limitation is the fact that it is not context-dependent. The set of rules we produced are at the "type level". It means that it does not take into account what are the instances of the UEML Base. A new set of rules could be produced. These rules would constraint how the meta-meta-model can be extended depending on which ontology phenomena the constructs are mapped. This work would enable one to improve the meta-meta-model in terms of precision and formality.

"UEML Validator" could also be improved in the domain of *Transformation*. Indeed, it would be interesting to know if the *StateLaws* are consistent (i.e., if the state on which they act have properties that are compatible with the law). It could also check if all *Transformation* are possible, if all *Transformation* pre-state are possible (i.e., if it is possible to find a state instance compliant with the pre-state definition) and if all *Transformation* post-states are consistent (i.e., will they still be compliant with the *stateLaws*?). An interesting way to achieve these objectives would be to use Alloy [All].

The "UEML Validator" graphical interface is really basic and could be improved. Checking the whole base can take a long time. Showing what is happening during the checking would be more convenient. It could also offer the possibility to select the rules to be checked directly in the program.

Studying new languages is another important need. Now that the method and the main tools are completed, we need to study the languages in order to have a broad base of knowledge. This base will allow comparison of these language semantics, to make model to model comparison or to perform language update reflection.

It would also be interesting to create a tool that could find the constructs representing the same kind of things. This tool would reveal the languages having constructs set representing the same part of reality. It could graphically show what part of the world the analysed EMLs represent and which ones represent which parts.

Bibliography

[All] Alloy.URL:http://alloy.mit.edu/.

[BAO04] Giuseppe Berio, Víctor Anaya, and Angel Ortiz. Supporting Enterprise Integration through a Unified Enterprise Modeling Language. *Conference on Advanced Information Systems Engineering*, 2004.

[Ber03] Giuseppe Berio. Deliverable D3.1: Requirements analysis: initial core constructs and architecture). may 2003.

[Ber05a] Giuseppe Berio. UEML 1.0 and UEML 2.0: Benefits, Problems and Comparison. In *Business Process Management Workshops*, pages 245–256, 2005.

[Ber05b] Giuseppe Berio. UEML 2.0 Deliverable 5.1. *INTEROP project UE-IST-508011*, 2005.

[Ber06] Giuseppe Berio. DEM project presentation, bergen, May 2006. URL: www.interop-noe.org.

[BPP04] Giuseppe Berio, Herve Panetto, and Michaël Petit. UEML: résultats et enjeux d'un langage unifié de modélisation d'entreprise. MOSIM'04, Nantes (France), September 2004.

[Bun77] Mario Bunge. *Treatise on Basic Philosophy, Volume 3*, chapter Ontology I: The Furniture of the World. Reidel, Boston, 1977.

[Bun79] Mario Bunge. *Treatise on Basic Philosophy, Volume 4*, chapter Ontology II: A World of Systems. Reidel, Boston, 1979.

[Chi96] Roderick Chisholm. *A Realistic Theory of Categories: An Essay on Ontology*. Cambridge University Press, 1996.

[Cor] Raul Corazzon. Ontology: A ressource guide for Pholosophers. URL: http://www.formalontology.it.

[DHP05] Gautier Dallons, Patrick Heymans, and Isabelle Pollet. A template-based analysis of GRL. In Proceedings of the 10th Int. Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD05), pages 493–504, Porto, June 2005.

[DSB+04] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. *OWL Web Ontology Language - Reference*. W3C Recommendation, http://www.w3.org/TR/2004/REC-owl-ref-20040210, 2004. Latest version available at http://www.w3.org/TR/owl-ref/.

[DVC98] Guy Doumeingts, Bruno Vallespir, and David Chen. *Handbook on architecture for Information Systems*, chapter Decision modelling GRAI grid. Springer-Verlag, 1998.

[DVZC92] Guy Doumeingts, Bruno Vallespir, M Zanettin, and David Chen. Gim, grai integrated methodology, a methodology for designing cim systems, version 1.0, unnumbered report. Technical report, University of Bordeaux 1, 1992.

[EEM] External, Extended Enterprise Ressources, Networks and Learnings, EC Project, IST-1999-10091, 2000.

[Gre96] P.F. Green. *An Ontological Analysis of Information Systems Analysis and Design (ISAD) Grammars in Upper CASE Tools*. PhD thesis, Department of Commerce, University of Queensland, 1996.

[ITU03] ITU. Recommendation Z.151 (GRL) version 3.0, September 2003.

[KBB03] Thomas Knothe, Christian Busselt, and Dieter Bll. Deliverable D2.3: Report on UEML (Needs and Requirements). 2003.

[KLS] John Krogstie, Odd Ivar Lindland, and Guttorm Sindre. Defining quality aspects for conceptual models. In *E. D. Falkenberg, W. Hesse, and A. Olive (Eds.)*, Marburg, Germany, March 28-30. Proceedings of the IFIP8.1 working conference on Information Systems Concepts (ISCO3); Towards a consolidation of views.

[Lam03] Axel Van Lamsweerde. The kaos meta-model: Ten years after. Technical report, Univeristé Catholique de Louvain, 2003.

[Let01] Emmanuel Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Université Catholique de Louvain, 2001.

[Lib] Jena Librairie.URL:http://jena.sourceforge.net/.

[Lp] Java Prolog Library.URL:http://www.swi-prolog.org/packages/jpl/.

[MHO06] Raimundas Matulevicius, Patrick Heymans, and Andreas L. Opdahl. Comparing GRL and KAOS using the UEML approach. In Proceedings of the 2nd International I-ESA 2006 Workshop on Enterprise Integration, Interoperability and Networking (EI2N 2006), Bordeaux, France, March 2006.

[MJ99] Kai Mertins and Roland Jochem. *Quality-Oriented Design of Business Processes*. Kluwer Academic Publishers, Boston/Dodrecht/London, 1999. ISBN 0-7923-8484-9.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*. W3C Recommendation, http://www.w3.org/TR/2004/REC-owl-features-20040210, 10 February 2004. Latest version available at http://www.w3.org/TR/owl-features/.

[OA05] Angel Ortiz and Víctor Anaya. Criteria to select EMLs. *INTEROP project UE-IST-508011*, 2005.

[oAfEI99] IFIP-IFAC Task Force on Architectures for Enterprise Integration. Geram: Generalised enterprise reference architecture and methodology. Technical Report Version 1.6.3, http://www.cit.gu.edu.au/ bernus/taskforce/geram/versions/, March 1999.

[OHS04] Andreas L. Opdahl and Brian Henderson-Sellers. A template for defining enterprise modelling constructs. *Journal of Database Management (JDM) 15(1)*, 2004.

[OHS05] Andreas L. Opdahl and Brian Henderson-Sellers. *Template-Based Definition of Information Systems and Enterprise Modelling Constructs*, chapter 6 in Ontologies and Business System Analysis, Peter Green and Michael Rosemann (eds.). Idea Group Publishing, 2005.

[OMG05] Object Management Group OMG. OCL 2.0 Specification. 2005.

[Opd05] Andreas L. Opdahl. UEML template version 1.1. *WP5*, 2005.

[Opd06] Andreas L. Opdahl. The UEML Approach to Modelling Construct Description. In Proceedings of I-ESA'06, Bordeaux (France), March 2006.

[PD02] Michaël Petit and Guy Doumeingts. Deliverable d1.1 : Report on the state of the art in enterprise modelling. 2002.

[Pro] Protégé.URL:http://protege.stanford.edu/.

[PS04] Peter F. Patel-Schneider. What is OWL (and why should I care)? Invited paper for the Ninth International Conference on the Principles of Knowledge Representation and Reasoning, Whistler, Canada, June 2004.

[PW97] J. Parsons and Yair Wand. Using objects for systems analysis. *Communication of the ACM*, 1997.

[UEMa] UEML1.0. URL: http://www.UEML.org.

[UEMb] UEML2.0. URL: http://www.interop-noe.org.

[UKM98] Uschold, King, and Moralee. The enterprise ontology. *The Knowledge Engeneering Review 13*, 1998.

[Ver96] Françcois B. Vernadat. *Enterprise modeling and integration: principles and applications*. Chapman and Hall, 1996.

[VS01] Verrijn and Stuart. A Framework of Information System Concepts - the revised FRISCO report. Web Document, 2001. draft version.

[Wae03] A. Walker and al. (eds.). *The New International Websters Comprehensive Dictionary of the English Language*. Encyclopaedic edition, Triden Press International, 2003.

[Web97] Ron Weber. Ontological foundation of information systems. In *Number 4*, Accounting Research Methodology Monograph. Coopers and Lybrand, 333, Collins Street, Melbourne Vic 3000, Australia, 1997.

[Wik] Wikipedia. Ontology (computer science). URL: http://en.wikipedia.org/wiki/Ontology_%28computer_science%29.

[WW88] Yair Wand and Ron Weber. An ontological analysis of some fundamental information systems concepts. Ninth International Conference on Information Systems, 30 December 1988.

[WW93] Yair Wand and Ron Weber. On the ontological expressiveness of information systems analysis and design grammars. *Journal of Information Systems*, pages 3:217–237, 1993.

[WW95] Yair Wand and Ron Weber. On the deep structure of informations systems. *Journal of Information Systems*, pages 5:203–223, 1995.

[WZ96] Ron Weber and Yair Zhang. An analytical evaluation of niam's grammar for conceptual schema diagrams. *Information Systems Journal*, pages 6:147–170, 1996.

[Yu97] Eric Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Symposium on Requirements Engineering (RE97)*. IEEE Computer Society, 1997.

Part IV

Appendix

Appendix A

Implemented Constraints

A.1 Mandatory fields

In this section we give the list of mandatory attributes for each entity. "Validator" show an error if one of this field was empty. We only give the OCL expression for the first one because the other are really similar.

LanguageDescription

1. languageName
2. languageVersion

Context: LanguageDescription
inv: languageName.notEmpty() AND languageVersion.notEmpty()

DiagramTypeDescription

3. diagramTypeName

Context: DiagramTypeDescription
inv: diagramTypeName.notEmpty()

ConstructDescription

4. constructName
5. instantiationLevel

Context: ConstructDescription
inv: constructName.notEmpty() AND instantiationLevel.notEmpty()

RepresentedPhenomenon

6. TypeOrValueOrNot
7. RoleName

Context: RepresentedPhenomenon
inv: TypeOrValueOrNot.notEmpty() AND RoleName.notEmpty()

OntologyPhenomenon

8. Name

Context: OntologyPhenomenon
inv: Name.notEmpty()

A.2 Relationships between entities

In this section we list the constraints concerning direct relationships between entities. Those constraints are a translation of the cardinalities we can find on the meta-meta-model. They have to be checked by "Validator" because there is nothing to check it within OWL. As for the previous section, OCL expression is not given for every constraint for more clarity.

9. Each *DiagramTypeDescription* has to be defined by a language.

Context: DiagramTypeDescription
inv: definedByLanguage.notEmpty()

10. Each *ConstructDescription* has to belong to a language;
11. and has to be used by a *diagramTypeDescription*;
12. defined by the same language.

Context: ConstructDescription
inv: belongsToLanguage.notEmpty() AND
usedInDiagram.notEmpty() AND
belongsToLanguage =
usesDescription.definedByLanguage

13. Each *relatedConstructName* of a *ConstructDescription* have to be another *ConstructDescription* belonging to the same language.

Context: ConstructDescription
inv: belongsToLanguage.definesConstruct $->$ includes(relatedConstructName)

14. Each *ConstructDescription* has to be described by at least *representedPhenonmenon*.

Context: ConstructDescription
inv: describedBy.notEmpty()

15. Each *RepresentedPhenomenon* has to represent an *OntPhenomenon*.

Context: RepresentedPhenomenon
inv: represents.notEmpty()

16. Each *RepresentedProperty* must characterize a *RepresentedClass*.

Context: RepresentedProperty
inv: characterics.notEmpty()

17. Each *RepresentedState* must be defined by a *RepresentedProperty*.

Context: RepresentedStateLaw
inv: RestrictsState.notEmpty()

18. Each *RepresentedTransformation* must have a pre state.

19. Each *RepresentedTransformation* must have a post state.

Context: RepresentedTransformation
inv: preRepState.notEmpty() AND
postRepState.notEmpty()

20. Each *RepresentedStateLaw* must restrict a *RespresentedState*.

Context: RepresentedStateLaw
inv: RestrictsState.notEmpty()

21. Each *RepresentedTransformationLaw* must effects a *RepresentedTransformation*.

Context: RepresentedStateLaw
inv: effectsRepTransformation.notEmpty()

22. Each *RepresentedPhenomenon* must represent a *ConstructDescription*.

Context: RepresentedPhenomenon
inv: describesConstruct.notEmpty()

23. Each *OntClass* has to be characterized by at least one *ontProperty*.

Context: OntClass
inv: Characterized.notEmpty()

24. Each *ontProperty* has to characterize an *OntClass*.

Context: OntProperty
inv: characterizes.notEmpty()

25. Each *OntTransformation* must have a pre state.

Context: OntTransformation
inv: preState.notEmpty()

26. Each *OntTransformation* must have a post state.

Context: OntTransformation
inv: postState.notEmpty()

27. Each *OntStateLaw* must restrict an *OntState*.

Context: OntStateLaw
inv: restrictsState.notEmpty()

28. Each *OntState* must be defined by at least one *ontProperty*.

Context: OntState
inv: definedBy.notEmpty()

29. Each *OntTransformationLaw* must effect an *ontTransformation*.

Context: OntTransformationLaw
inv: effectsTransformation.notEmpty()

A.3 Identifying Names

In this section we give the list of names that have to be unique in a certain scope.

30. Two *ConstructDescriptions* cannot have the same *constructName* inside the same language.

Context: LanguageDescription
inv: definesConstruct− >isUnique(constructName)

31. The *RoleName* of each *RepresentedPhenomenon* of a *ConstructDescription* must be different.

Context: ConstructDescription
inv: describedBy− >isUnique(roleName)

32. Each *OntPhenomenon* must have a different name.

Context: OntPhenomenon
inv: isUnique(ontologyPhenomeonName)

33. If a *RepresentedClass* has more than one *RepresentedProperty* then each of them have to have a different roleName.

Context: RepresentedClass
Inv: self.characteristics $->$ *forAll*(p1,p2 |
p1 <> p2 *implies* p1.roleName <> p2.roleName)

A.4 Classes

In this section we produce the constraints concerning represented an ontological classes. They deal with uniqueness of *ontClasses* in term of *ontProperties* set and with the "generalization" and "represents" relationship.

34. Two different *OntClasses* cannot have the same sets of *OntologyProperties.*

Context: OntClass
inv: Self $->$ *forAll*(c1, c2 | c1 <> c2 *implies*
c1.characterizedBy <> c2.characterizedBy)

35. If the set of *OntProperty* of an *OntClass* is a subset of the set of *OntProperty* of another *OntClass*, then the first *OntClass* must generalize the second one.

    If all the properties of an *OntClass* are the one or precede the one of another class, then this class must generalize the other.

Context: OntClass
inv: self $->$ *forAll* (c1,c2 | c1.characterizedBy.includesAll(c2.characterizedBy)
implies c1.genClass(c2))
Context: OntClass
inv: self $->$ *forAll*(c1,c2 | c1.characterizedBy excludes(c2.characterizedBy)
$->$ *forAll*(e | e.precedes(c1.X) *implies* c1.genClass(c2)))

36. There cannot have any cycle in the generalization relationship.

Context: OntClass
inv: **let** generalize(X) : Boolean = self.genClass(X)
generalize(X) : Boolean = not(self.genClass(X)) and
Z.genClass(X) and self.generalize(Z)
in: self.genClass $->$ *forAll*(c1 | c1.generalize.excludes(self))

37. A *RepresentedClass* must represents an *OntClass.*

Context: representedClass
inv: represents.notEmpty()

A.5 Properties

It this section the constraints concerning Properties are listed. They deal with the *RepresentedClass* and *RepresentedProperties* a *constructDescription* have to possess, the coherence between *Properties* sets of *RepresentedClass* and *OntClass*, the coherence between *Classes* of *RepresentedProperties* and *OntProperties*, the "precedence" relationship, the coherence of the "contain" relationship, the *StateLawProperties*, the *TransformationLawProperties* and with the "represent" relationship.

38. A *ConstructDescription* must describe the *RepresentedProperties* of the *RepresentedClass* it describes;

39. and vice-versa.

Context: ConstructDescription
Inv: describedBy− >select(c: RepresentedClass | c.characteristics.describesConstruct.includes(self))
and
self.describedBy− >select(c: RepresentedProperty | c.characteristics.describesConstruct.includes(self))

40. If a *RepresentedClass* has a *RepresentedProperty* then the corresponding *OntClass* must have the corresponding *OntProperty*.

Context: RepresentedClass
Inv: represents.characteristics.includesAll(self.characteristics.represents)

41. If a *RepresentedProperty* characterize a *RepresentedClass* then the corresponding *OntProperty* must characterize the corresponding *OntClass.*

Context: RepresentedProperty
Inv: represents.characteristics.includes(self.characteristics.represents)

42. An *OntologyProperty* cannot be preceded by itself.

Context: OntProperty
Inv: precededBy.excludes(self)

43. If an *OntClass* is characterized by an *OntProperty* that is preceded by another *OntProperty*, then it must also be characterized by the second *OntProperty.*

Context: OntClass
Inv: characteristics− >forAll(p | self.includesAll(p.preceded))

44. If an *OntProperty* is preceded by a second *OntProperty* and the second *OntProperty* is preceded by a third one, then the first *OntProperty* must also be preceded by the third one.

Context: OntProperty
Inv: precededBy− > $forAll$(p2 | p2.precededBy − >
$forAll$(p3 | p.precededBy(p3)))

45. All the sub-properties of an *OntComplexProperty* have to characterize either the *OntClass* that the *OntComplexProperty* characterize or a *super-OntClass* of *OntClass* that the *complex-OntProperty* characterize.

Context: OntProperty
Inv: contains $->forAll$(sp | sp.characteristics.excludes(self.characteristics)
$->forAll$(sp2 | sp2.characteristics.genClass.includes(self.characteristics)))

46. Each sub-Property has to be preceded by its complex-Property.

Context: OntProperty
Inv: contains $->forAll$(sp | sp.precededBy.includes(self)

47. An *OntProperty* cannot contain another *OntProperty* that already contains it.

Context: OntProperty
Inv: containsProp.excludes(self.containedByProp)

48. An *OntProperty* cannot precede another *OntProperty* that already precedes it.

Context: OntProperty
Inv: precedes.excludes(self.precededBy)

49. The *OntProperties* defining an *OntState* on which an *OntStateLaw* acts must be contained by this *OntStateLaw*.

Context: OntProperty
Inv: containedBy.includes(self.defining.restrictedByStateLaw)

50. The *OntProperties* defining an *OntState* that is pre- or post-State of an *OntTransformation* must be sub-Properties of the *OntTransformationLaw* that effects the *OntTransformation* (if there is one).

Context: OntProperty
Inv: containedBy.includes(self.defining.entryTransformation.effectedByTransformation)
AND
containedBy.includes(self.defining.exitTransformation.effectedByTransformation)

51. A *RepresentedProperty* must represent an *OntProperty*.

52. A *RepresentedTransformationLaw* must represent an *OntTransformationLaw*.

53. A *RepresentedStateLaw* must represent an *OntStateLaw*.

A.6 States

The constraints concerning States are concentrated in this section. Those constraints deal with the *Class* of *State*, the *Properties* of a *State*, and with the "generalization" and "represent" relationships.

54. A *RepresentedState* must be restricted by a *RepresentedStateLawProperty*.

Context: RepresentedState
Inv: restrictedByStateLaw.notEmpty()

55. If a *RepresentedState* has a set of *RepresentedProperties*, there must be a *RepresentedClass* whose set of characteristic *RepresentedProperties* is a (possibly improper) superset of the first set.

Context: RepresentedState
Inv: self $->$ $forAll$(rs | RepresentedClass $->$ $exists$(c |
c.characteristics.includesAll(rs.defining)))

56. If an *OntState* has a set *OntProperties*, there must be a *OntClass* whose set of characteristic *OntProperties* is a (possibly improper) superset of the first set.

Context: OntState
Inv: self $->$ $forAll$(rs | OntologyClass $->$ $exists$(c |
c.characteristics.includesAll(rs.defining)))

57. If a *ConstructDescription* contains a *RepresentedState* and the corresponding *OntState* has a set of *OntProperties*, then there must be an *OntClass* whose set of characteristic *OntProperties* is a (possibly improper) superset of the first set and the *ConstructDescription* must contain the corresponding *RepresentedClass.*

Context: ConstructDescription
Inv: describedBy$->$select(rs : RepresentedState |
self.includes(RepresentedClass c |
c.represents.characteristics.includesAll(rs.defining)))

58. If a *ConstructDescription* contains a *RepresentedState* and the corresponding *OntState* has an *OntProperty*, then the *ConstructDescription* must also contain a corresponding *RepresentedProperty.*

Context: ConstructDescription
Inv: describedBy$->$select(rs : RepresentedState | self.includes(rs.defining))

59. If an *OntState* generalize another *OntState* then all the properties it defines must also be defined by the other state.

Context: OntState
Inv: defining.includesAll(self.specState.defining)

60. An *OntState* cannot specialize an *OntState* that already specialize it.

Context: OntState
inv: **let** generalize(X) : Boolean = self.genState(X)
generalize(X) : Boolean = not(self.genState(X)) and
Z.genState(X) and self.generalize(Z)
in: genState $->$ $forAll$(s1 | s1.generalize.excludes(self))

61. A *RepresentedState* must represent an *OntState.*

A.7 Transformations

The constraints concerning Transformations ensure that two *OntTransformation* are not the same in term of from- and to-*State* and the coherence of the "contains" and "represents" relationships.

62. Two distinct *OntTransformations* cannot have identical from- and to-States.

Context: OntTransformation
Inv: self $->$ $forAll$(t1,t2 | t1.preState=t2.preState *and*
t1.toState=t2.toState *implies* t1=t2)

63. A *RepresentedTransformation* must represent an *OntTransformation.*

64. An *OntTransformation* cannot contain an *OntTransformation* that already contains it.

Context: OntTransformation
inv: **let** generalize(X) : Boolean = self.genTransf(X)
generalize(X) : Boolean = not(self.genTransf(X)) and
Z.genState(X) and self.generalize(Z)
in: genTransf $->$ $forAll$(s1 | s1.generalize.excludes(self))

A.8 Several ConstructDescriptions

65. A *representedPhenomenon* can only be described by one construct.

Context: RepresentedPhenomenon
Inv: describedBy.size() = 1

Appendix B

Constraints in Prolog

```
%%%%%%%%%%%%%%%%%%%%%%% Intermediate definitions %%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% True if the class X has the property A or a property that precedes A.
hasProperty(X,A):- ontologyclass(X), relationtoproperty(X,Y),
        property(Y,A).

hasProperty(X,A):- ontologyclass(X), relationtoproperty(X,Y),
        property(Y,B),precedes(A,B).

%%%%%%%%% Y is the property of X
haveProp(X,Y):- ontologyclass(X),ontologyproperty(Y),
        relationtoproperty(X,Z),property(Z,Y).

%%%%%%%%% Y is the RepProperty of X
haveRepProp(X,Y):- representedclass(X),representedproperty(Y),
        relationtoreppproperty(X,Z),reppproperty(Z,Y).

%%%%%%%%%% X is the superClass of Y
ontClassGeneralize(X,Y):- ontologyclass(X), ontologyclass(Y),
        specialisationrelation(X,A), subclass(A,Y).

%%%%%%%%%% X is a property of Y
belongsToClass(X,Y):- ontologyproperty(X),ontologyclass(Y),
        relationtoclass(X,Z),class(Z,Y).

%%%%%%%%%% SP is a subProperty of the complexProp Y
subAndComplex(SP,CP):- ontologyproperty(SP), ontologyproperty(CP),
        relationtosubproperty(CP,X), subproperty(X,SP).

%%%%%%%%%% The porperty X contains the property Y
containsProp(X,Y):- relationtosuperproperty(X,A), superproperty(A,Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Rules%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%% About mandatory fields %%%%%%%%%%%%%%%%%%%%%%%%%%%

%1%%%% language without Name
languageWithoutName(X):- languagedescription(X),
        not(languagedescription-languagename(X,_)).

%2%%%% language without Version
languageWithoutVersion(X):- languagedescription(X),
```

```
        not(languagedescription-languageversion(X,_)).

%3%%%% DiagramTypeDescription
diagWithoutName(X):- diagramtypedescription(X),
        not(diagramtypedescription-diagramtypename(X,_)).

%4%%%% constructDescription - Name
constructDescriptionWithoutName(X):- constructdescription(X),
        not(constructdescription-constructname(X,_)).

%5%%%% constructDescription - Instanciation-level
constructDescriptionWithoutInstLevel(X):- constructdescription(X),
        not(instantiationlevel(X,_)).

%6%%%% Each RepProp must be TypeOrValueOrNot
reprPropertyWithoutTypeOrValue(X):- representedproperty(X),
        not(istypeorvalueornot(X,_)).

%7%%%% Each RepPhenomena must have a role-name
repPhenomenonWithoutName(X):- namedrepresentedphenomenon(X),
        not(namedrepresentedphenomenon-rolename(X,_)).

%8%%%% Each OntPhenomena must have a name
ontPhenomenonWithoutName(X):- namedontologyphenomenon(X),
        not(namedontologyphenomenon-name(X,_)).

%%%%%%%%%%%%%%%%%%%% Relationships - Cardinalities %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% Language - Constructs

%9%%%% relationshipofDiagramTypeDescription
definedBylanguage(X):- diagramtypedescription(X),languagedescription(Y),
        definesdiagramtype(Y,X).

diagramWithoutLanguage(X):- diagramtypedescription(X),
        not(definedBylanguage(X)).

%10%%%% A construct has to belong to a language.
constructDescriptionOfLanguage(X):- constructdescription(X),
        languagedescription(Y), definesconstruct(Y,X).

badConstructDescriptionOfLanguage(X):-constructdescription(X),
        not(constructDescriptionOfLanguage(X)).

%11%%%% A Construct has to be used by a Diagram.
badConstructDescriptionofDiagramTypeDescr(X):- constructdescription(X),
        not(usesconstruct(_,X)).

%12%%%% A Construct has to be used by a Diagram of the same Language.
constructOfBadDiagram(X,Diag,LgCons,LgDiag):- constructdescription(X),
        belongstolanguage(X,LgCons), usesconstruct(Diag,X),
        definedbylanguage(Diag,LgDiag), not(LgCons=LgDiag).

%13%%%% relatedConstructName
constructOfSameDiagram(X,Y):- constructdescription(X),
        constructdescription(Y), diagramtypedescription(Z),
        usesconstruct(Z,X), usesconstruct(Z,Y).

relatedConstructOK(X,Y):- constructdescription(X),
```

```
        constructdescription-relatedconstructnames(X,Y),
        constructdescription(Y), constructOfSameDiagram(X,Y).

relatedConstructKO(X,Y):- constructdescription(X),
        constructdescription-relatedconstructnames(X,Y),
        not(constructdescription(Y)).

relatedConstructKO(X,Y):- constructdescription(X), constructdescription(Y),
        constructdescription-relatedconstructnames(X,Y),
        not(constructOfSameDiagram(X,Y)).

%14%%%% Each ConstructDescr has to be described by at least repPhen
constructWithoutRepPhen(X):- constructdescription(X),
        not(describedby(X,_)).

%%%%%%%%% RepPhenomenon

%15%%%% Each "representedPhenomena" must represents an "ontPhenomena"
representedPhenomeonOK(X):- namedrepresentedphenomenon(X),
        representedontologyphenomenon(Y), represents(X,Y).

badReprentedPhenomenon(X):- namedrepresentedphenomenon(X),
        not(representedPhenomeonOK(X)).

%16%%%% a representedProperty must characterise a repClass
badRepresentedProperty(X):- representedproperty(X),
        not(reltoclass(X)).

reltoclass(X):- repproperty(Y,X), repclass(Y,_).

%17%%%% a representedState must be defined by a RepProperty
representedStateWithoutRepProp(X):- representedstate(X),
        not(definedbyrepproperty(X,_)).

%18%%%% Each RepresentedTransformation must have pre state
representedTransformationWithoutPre(X):- representedtransformation(X),
        not(prerepstate(X,_)).

%19%%%% Each RepresentedTransformation must have post state
representedTransformationWithoutPost(X):- representedtransformation(X),
        not(postrepstate(X,_)).

%20%%%% Each "RepresentedStateLaw" must restrict a "RespresentedState"
badRepresentedStateLaw(X):- representedstatelaw(X),
         not(restrictsrepstate(X,_)).

%21%%%% a reprTransfLaw must effects a reprTransf.
reprTransfLawWithoutTransf(X):- representedtransformationlaw(X),
        not(effectsreptransformation(X,_)).

%22%%%% a RepresentedPhenomena must represent a Construct
repPhenWithoutConstruct(X):- namedrepresentedphenomenon(X),
        not(describesconstruct(X,_)).

%%%%%%%%% OntPhenomenon

%23%%%% Each OntologyClass has to be characterized by at least a ontPropery
ontologyClassOK(X):- ontologyclass(X),
```

```
        ontologyclasspropertyrelation(Y), class(Y,X), property(Y,_).

badOntologyClass(X):- ontologyclass(X), not(ontologyClassOK(X)).

%24%%%% Each ontPropery has to characterize an OntologyClass
ontologyPropertyOK(X):- ontologyproperty(X),
        ontologyclasspropertyrelation(Y), class(Y,_), property(Y,X).

badOntologyProperty(X):- ontologyproperty(X), not(ontologyPropertyOK(X)).

%25%%%% Each OntologyTransformation must have pre state
ontologyTransformationWithoutPreState(X):- ontologytransformation(X),
        not(prestate(X,_)).

%26%%%% Each OntologyTransformation must have post state
ontologyTransformationWithoutPostState(X):- ontologytransformation(X),
        not(poststate(X,_)).

%27%%%% Each ontologyStateLaw must restrict an ontologyState
badOntologyStateLaw(X):- ontologystatelaw(X), not(restrictsstate(X,_)).

%28%%%% Each ontState must be defined by at least one ontProperty
ontStateLawWithoutProp(X):- ontologystate(X),
        not(definedbyproperty(X,_)).

%29%%%% Each ontTransfLaw must effects an ontTransf.
ontTransfLawWithoutTransf(X):- ontologytransformationlaw(X),
        not(effectstransformation(X,_)).

%%%%%%%%%%%%%%%%%%%%%%%% Constraints on names %%%%%%%%%%%%%%%%%%%%%%%%

%30%%%% constructName must be <> for every construct of a language
badNameofConstruct(X,Y,Z,A):-constructdescription(X),
        constructdescription(Y),not(X=Y),
        belongstolanguage(X,A), belongstolanguage(Z,A),
        constructdescription-constructname(X,Z),
        constructdescription-constructname(Y,Z).

%31%%%% uniqueness of Class roleNames within ConstructDefinitions
roleNameNotUnique(X,Y,Z):- constructdescription(X),
        namedrepresentedphenomenon(Y), describedby(X,Y),
        namedrepresentedphenomenon(Z), describedby(X,Z),
        not(Y = Z), namedrepresentedphenomenon-rolename(Z,A),
        namedrepresentedphenomenon-rolename(Y,B), (A=B).

%32%%%% Each OntologyPhenomena must have a different name
ontologyPhenomenaNotUnique(X,Y,A):- namedontologyphenomenon(X),
        namedontologyphenomenon(Y), not(X=Y),
        namedontologyphenomenon-name(X,A),
        namedontologyphenomenon-name(Y,B), (A=B).

%33%%%% If a RepClass has more than one RepProp, each of them must
%%%%%%% have a roleName that is unique relative to the RepClass.
sameRoleName(X,Y,Z,C):- representedclass(X), representedproperty(Y),
        relationtorepproperty(X,A),     repproperty(A,Y),
        representedproperty(Z), relationtorepproperty(X,B), not(A=B),
        repproperty(B,Z), representedproperty-rolename(Y,C),
        representedproperty-rolename(Z,D), (C=D).
```

```
%%%%%%%%%%% Constraints on Classes and RepresentedClasses %%%%%%%%%%%

%34%%%% Two different ontClasses cannot have the same sets of ontProp.
hasDifferentProp(X,Y):- ontologyclass(X), ontologyclass(Y), not(X=Y),
        relationtoproperty(X,A), property(A,C), relationtoproperty(Y,B),
        property(B,D), not(C=D).

sameProperties(X,Y):- ontologyclass(X), relationtoproperty(X,_),
        ontologyclass(Y), relationtoproperty(Y,_), not(X=Y),
        not(hasDifferentProp(X,Y)).

%35%%%% If the set of ontProperties of an ontClass is a subset of
%%%%%%% another ontClass, the first ontClass must generalise the second.
%%%%%%% If all the properties of an OntologyClass precede the one of another
%%%%%%% class, then this class must generalise the other.
noPropertyOfXInY(X,Y):- ontologyclass(X), ontologyclass(Y), not(X=Y),
        relationtoproperty(X,A), property(A,C), not(hasProperty(Y,C)).

doNotGeneralize(X,Y):- ontologyclass(X), ontologyclass(Y), not(X=Y),
        not(noPropertyOfXInY(X,Y)), not(ontClassGeneralize(X,Y)).

%36%%%% No cycle in with Generalization for Classes
specialize(X,Y):-specialisationrelation(X,A), superclass(A,Y).

generalize(X,Y):-specialisationrelation(X,A), subclass(A,Y).

classGeneralize(X,Y):- ontologyclass(X), ontologyclass(Y),
        generalize(X,Y).

classGeneralize(X,Y):- ontologyclass(X), ontologyclass(Y),
        not(generalize(X,Y)), generalize(X,Z),
        classGeneralize(Z,Y).

cylceInClassGeneralization(X,Y):- ontologyclass(X), ontologyclass(Y),
        specialize(X,Y),classGeneralize(X,Y).

%37%%%% Each reprClass must represent a OntClass
repClassNotReprOntClass(X):- representedclass(X), represents(X,Y),
        not(ontologyclass(Y)).

%%%%%%%%%% Constraints on OntProperties and RepresentedProperties %%%%%%%%%%%

%38%%%% a construct must containt the properties of the repClass.
hasAllProperties(X,A):- constructdescription(X), describedby(X,Y),
        relationtorepproperty(Y,Z),repproperty(Z,A), not(describedby(X,A)).

%39%%%% The reverse
hasAllClasses(X,A):- constructdescription(X), describedby(X,Y),
        relationtorepclass(Y,Z),repclass(Z,A), not(describedby(X,A)).

%40%%%% If a RepClass has a RepProp, the corresponding OntClass
%%%%%% must have the corresponding OntProp as characteristic.
isRepCharacterizedBy(Class,Prop):- representedclass(Class),
        representedproperty(Prop),relationtorepproperty(Class,X),
        repproperty(X,Prop).

notHasOntProp(I,J,K,L):- representedclass(I), represents(I,K),
```

```
        isRepCharacterizedBy(I,J), represents(J,L),
        not(haveProp(K,L)).

%41%%%% The reverse (if a RepProp belongs to a repClass.....)
notBelongsToOntClass(I,J,K,L):- representedproperty(I),
        represents(I,K), relationtorepclass(I,X), repclass(X,J),
        represents(J,L), not(belongsToClass(K,L)).

%42%%%% A Property cannot be preceded by itself.
badPrecedence(X):- ontologyproperty(X), precededby(X,X).

%43%%% If an ontClass has an ontProp that is preceded by another
%%%%%% ontProp, then it must also have the second ontProp.
precededPropTheClassShouldHave(A,B,C):- ontologyclass(A),
        relationtoproperty(A,X), property(X,B), precededby(B,C),
        not(haveProp(A,C)).

%44%%% If an ontProp is preceded by a second ontProp and the
%%% second ontProp is preceded by a third one, then the first
%%% ontProp must also be preceded by the third ontProp.
notHaveAllPrecedence(X,Y,Z):- ontologyproperty(X), precededby(X,Y),
        precededby(Y,Z), not(precededby(X,Z)).

%45%%%% All the subproperties of an ontologyComplexProperty have to
%%%%%%% characterise either the OntologyClass that the
%%%%%%% OntologyComplexProperty characterise or a subOntologyClass of
%%%%%%% OntologyClass that the complexOntologyProperty characterise.
  %B is the subclass of C%
sameOrSubClass(B,C):- ontologyclass(B), ontologyclass(C),
        generalisationrelation(B,X), superclass(X,C).

sameOrSubClass(B,C):- ontologyclass(B), ontologyclass(C),(B=C).

subPropOfClassOK(X) :- ontologyproperty(X), relationtosuperproperty(X,A),
        superproperty(A,Y), belongsToClass(X,B), belongsToClass(Y,C),
        sameOrSubClass(B,C).

subPropOfBadClass(X):- ontologyproperty(X), relationtosuperproperty(X,_),
        not(subPropOfClassOK(X)).

%46%%%% Each subProperty has to be preceded by its complexProperty.
subPropNotPrecededByCompl(Sub,Compl):- ontologyproperty(Sub),
        ontologyproperty(Compl), subAndComplex(Sub,Compl),
        not(precededby(Sub,Compl)).

%47%%%% There can't have any cycle in the containsProp relationship.
isContainedBy(X,Y):- ontologyproperty(X), ontologyproperty(Y),
        containsProp(Y,X).

isContainedBy(X,Y):- ontologyproperty(X), ontologyproperty(Y),
        not(containsProp(Y,X)), containsProp(Y,Z), isContainedBy(X,Z).

cylceInContainsProp(X,Y):-ontologyproperty(X), ontologyproperty(Y),
        containsProp(X,Y),isContainedBy(X,Y).

%48%%%% There can't have any cycle in the precedesProp relationship.
isPrecededby(X,Y):- ontologyproperty(X), ontologyproperty(Y),
        precedes(Y,X).
```

```
isPrecededby(X,Y):- ontologyproperty(X), ontologyproperty(Y),
        not(precedes(Y,X)), precedes(Y,Z), isPrecededby(X,Z).

cylceInPrecedesProp(X,Y):-ontologyproperty(X), ontologyproperty(Y),
        precedes(X,Y),isPrecededby(X,Y).

%49%%%% The OntologyProperties defining an OntologyState on which an
%%%%%%% OntologyStateLaw acts must be sub-properties of this
%%%%%%% OntologyStateLaw.
propShouldBeSubOfStateLaw(X,Y,State):- ontologystatelaw(Y),
        restrictsstate(Y,State), definedbyproperty(State,X),
        not(containsProp(X,Y)).

%50%%%% The OntologyProperties defining an OntologyState that is pre- or
%%%%%%% post-State of an OntologyTransformation must be sub-Properties of the
%%%%%%% OntologyTransformationLaw that effects the OntologyTransformation
propShouldBeSubOfTransfLaw(Prop,TransLaw,Transf,State):-
        ontologytransformationlaw(TransLaw),
        effectstransformation(TransLaw,Transf), prestate(Transf,State),
        definedbyproperty(State,Prop),not(containsProp(TransLaw,Prop)).

propShouldBeSubOfTransfLaw(Prop,TransLaw,Transf,State):-
        ontologytransformationlaw(TransLaw),
        effectstransformation(TransLaw,Transf), poststate(Transf,State),
        definedbyproperty(State,Prop),not(containsProp(TransLaw,Prop)).

%51%%%% Each repProp must represent a OntProp
repPropNotReprOntProp(X):- representedproperty(X),
        not(representedtransformationlaw(X)),
        not(representedstatelaw(X)), represents(X,Y),
        not(ontologyproperty(Y)).

repPropNotReprOntProp(X):- representedproperty(X),
        not(representedtransformationlaw(X)),
        not(representedstatelaw(X)), represents(X,Y),
        ontologytransformationlaw(Y).

repPropNotReprOntProp(X):- representedproperty(X),
        not(representedtransformationlaw(X)),
        not(representedstatelaw(X)), represents(X,Y),
        ontologystatelaw(Y).

%52%%%% Each reprTransfLaw must represent a OntTransFormLaw
repTransfLawNotReprOntTransfLaw(X):- representedtransformationlaw(X),
        represents(X,Y), not(ontologytransformationlaw(Y)).

%53%%%% Each reprStateLaw must represent a OntStateLaw
repStateLawNotReprOntStateLaw(X):- representedstatelaw(X),
        represents(X,Y), not(ontologystatelaw(Y)).

%%%%%%%%%%%%%%%%%%%%%%%%%% Constraints on states %%%%%%%%%%%%%%%%%%%%%%%%%%

%54%%%% Each RepresentedState must be restricted by a RepresentedStateLawProperty
stateNotRestricted(X):-representedstate(X),
        not(restrictedbyrepstatelaw(X,_)).

%55%%%% A represented state must be the one of a class.
```

```
390  hasAnotherProp(X,Y):- representedstate(X), definedbyrepproperty(X,Z),
             not(Z=Y).

     stateOfOneProperty(X):- representedstate(X), definedbyrepproperty(X,Y),
             not(hasAnotherProp(X,Y)).
395
     repPropNotOfThisClass(Class,X):- representedclass(Class), representedstate(X),
             definedbyrepproperty(X,Y), not(haveRepProp(Class,Y)).

     classOfRepState(X,Class):- representedstate(X), not(stateOfOneProperty(X)),
400          definedbyrepproperty(X,Prop), !, haveRepProp(Class,Prop),
             not(repPropNotOfThisClass(Class,X)).

     stateOfNoClass(X):- representedstate(X), not(stateOfOneProperty(X)),
             not(classOfRepState(X,_)).
405
     %56%%%% An Ontology state must be the one of a class.
     hasAnotherOntProp(X,Y):- ontologystate(X), definedbyproperty(X,Z), not(Z=Y).

     stateOfOneOntProperty(X):- ontologystate(X), definedbyproperty(X,Y),
410          not(hasAnotherOntProp(X,Y)).

     propNotOfThisClass(Class,X):- ontologyclass(Class), ontologystate(X),
             definedbyproperty(X,Y), not(haveProp(Class,Y)).

415  classOfState(X,Class):- ontologystate(X), not(stateOfOneOntProperty(X)),
             definedbyproperty(X,Prop), !, haveProp(Class,Prop),
             not(propNotOfThisClass(Class,X)).

     noClassOfState(X):- ontologystate(X), not(stateOfOneOntProperty(X)),
420          not(classOfState(X,_)).

     %57%%%% if a constructDescr contains a state, then it must also contain
     %%%%%%% the class having the set of properties on wich the state acts
     %hasRepProp(C,P):- representedclass(C),representedproperty(P),
425  %       relationtorepproperty(C,Z),repproperty(Z,P).

     %propOfStateNotOfClass(P,C,S):- representedproperty(P), representedclass(C),
     %       representedstate(S), definedbyrepproperty(S,P),
     %       not(hasRepProp(C,P)).
430
     %classOfState(C,S):- representedclass(C), representedstate(S),
             not(propOfStateNotOfClass(_,C,S)).

     classOfStateButNotOfConstruct(Class,S,Construct):- representedclass(Class),
435          representedstate(S), not(stateOfOneProperty(S)), classOfRepState(S,Class),
             describedby(Construct,S), not(describedby(Construct,Class)).

     classOfStateButNotOfConstruct(Class,S,Construct):- representedclass(Class),
             representedstate(S), stateOfOneProperty(S), definedbyrepproperty(S,P),
440          haveRepProp(Class,P), describedby(Construct,S),
             not(describedby(Construct,Class)).

     %58%%%% If a Construct contains a ReprState and the corresponding
     %%%%%%% OntState has a ontProp, then the Construct must also
445  %%%%%%% contain a corresponding ReprProp
     constructWithoutPropOfState(A,B,D):- constructdescription(A),
             describedby(A,B), representedstate(B), represents(B,C),
```

```
        definedbyproperty(C,D), not(hasARepPropDescribedByConstr(D,A)).

hasARepPropDescribedByConstr(D,A):- ontologyproperty(D),
        constructdescription(A), representedby(D,X),
        describesconstruct(X,A).

%59%%%% A state that generalize another has to define all the property the
%%%%%%% other state define.
%properties that X don't define but that Y does.
nothasTheProperties(X,Y,A):- ontologystate(X), ontologystate(Y),
        definedbyproperty(Y,A), not(definedbyproperty(X,A)).

superStateWithPropMissing(X,Y,A):- ontologystate(X), ontologystate(Y),
        containsstate(X,Y), nothasTheProperties(X,Y,A).

%60%%%% No cycle with specializations for States
stateGeneralize(X,Y):- ontologystate(X), ontologystate(Y),
        containedbystate(X,Y).

stateGeneralize(X,Y):- ontologystate(X), ontologystate(Y),
        not(containedbystate(X,Y)), containsstate(X,Z),
        stateGeneralize(Z,Y).

cylceInStateGeneralization(X,Y):- ontologystate(X), ontologystate(Y),
        containsstate(X,Y), stateGeneralize(X,Y).

%61%%%% Each reprState must represent a OntState
repStateNotReprOntState(X):- representedstate(X),
        represents(X,Y), not(ontologystate(Y)).

%%%%%%%%%%%%%%%%%%%%%% Constraints on Transformations %%%%%%%%%%%%%%%%%%%%%%

%62%%%% 2 distinct ontTransf cannot have identical pre- & post States.
sameTransf(X,Y):- ontologytransformation(X),
        ontologytransformation(Y), not(Y=X), prestate(X,A),
        prestate(Y,B),(A=B),poststate(X,C),poststate(Y,D),(C=D).

%63%%%% Each reprTransf must represent a OntTransf
repTransfNotReprOntTransf(X):- representedtransformation(X),
        represents(X,Y), not(ontologytransformation(Y)).

%64%%%% No cycle with the contains/containedBy for Transformations
transIsContainedBy(X,Y):- ontologytransformation(X), ontologytransformation(Y),
        containstransformation(Y,X).

transIsContainedBy(X,Y):- ontologytransformation(X), ontologytransformation(Y),
        not(containstransformation(Y,X)), containstransformation(Y,Z),
        transIsContainedBy(X,Z).

cylceInContainsTransf(X,Y):- ontologytransformation(X), ontologytransformation(Y),
        containstransformation(X,Y),transIsContainedBy(X,Y).

%%%%%%%%%%%%%%%% About several ConstructDescriptions %%%%%%%%%%%%%%%%%%%%

%65%%%% A representedPhenomena can only describe one construct
repPhenomenaDiscribingSeveralConstr(X):- representedphenomenon(X),
        describesconstruct(X,Y), describesconstruct(X,Z), not(Y=Z).
```

510

Appendix C

Generated Fact Base

The following excerpt is a little part of the facts generated from the UEML Base 0.06 by UEML Validator.

```
wholeorpart(ispart).
wholeorpart(iswhole).
diagramtypedescription(umlclassdiagram).
diagramtypedescription-diagramtypename(umlclassdiagram,classdiagram).
definedbylanguage(umlclassdiagram,uml20).
usesconstruct(umlclassdiagram,umlassociation).
usesconstruct(umlclassdiagram,umlattribute).
usesconstruct(umlclassdiagram,umlobject).
usesconstruct(umlclassdiagram,umlclass).
usesconstruct(umlclassdiagram,umlaggregation).
usesconstruct(umlclassdiagram,umlcomposition).
usesconstruct(umlclassdiagram,umlproperty).
usesconstruct(umlclassdiagram,umlgeneralization).
ontologyproperty(anyregularproperty).
ontologyproperty-name(anyregularproperty,anyregularproperty).
representedby(anyregularproperty,umlobjectpropertyroleproperty).
representedby(anyregularproperty,umlattributepropertyroleattribute).
representedby(anyregularproperty,umlpropertypropertyroleproperty).
representedby(anyregularproperty,umlclasspropertyroleattribute).
relationtoclass(anyregularproperty,attributedthingsanyregularproperty).
ontologyproperty(partwholerelation).
precedes(partwholerelation,systempartwholerelation).
ontologyproperty-name(partwholerelation,partwholerelation).
representedby(partwholerelation,umlaggregationpropertyrolepartwholerelation).
precededby(partwholerelation,anymutualproperty).
relationtoclass(partwholerelation,componentthingspartwholerelation).
relationtoclass(partwholerelation,compositethingspartwholerelation).
ontologyproperty(systempartwholerelation).
ontologyproperty-name(systempartwholerelation,systempartwholerelation).
representedby(systempartwholerelation,umlcompositionpropertyrolepartwholerelation).
precededby(systempartwholerelation,partwholerelation).
relationtoclass(systempartwholerelation,systemthingssystempartwholerelation).
relationtoclass(systempartwholerelation,systemcomponentthingssystempartwholerelation).
ontologyproperty(abilitytoassociate).
precedes(abilitytoassociate,anymutualproperty).
precedes(abilitytoassociate,classsubclassrelationship).
ontologyproperty-name(abilitytoassociate,abilitytoassociate).
relationtoclass(abilitytoassociate,allthingsabilitytoassociate).
ontologyproperty(classsubclassrelationship).
```

```
ontologyproperty-name(classsubclassrelationship,classsubclassrelationship).
representedby(classsubclassrelationship,umlgeneralizationpropertyroleclasssubclassrelationship).
precededby(classsubclassrelationship,abilitytoassociate).
relationtoclass(classsubclassrelationship,allthingsclasssubclassrelationshipparent).
relationtoclass(classsubclassrelationship,allthingsclasssubclassrelationshipchild).
ontologyproperty(anymutualproperty).
precedes(anymutualproperty,partwholerelation).
ontologyproperty-name(anymutualproperty,anymutualproperty).
representedby(anymutualproperty,umlobjectpropertyrolelink).
representedby(anymutualproperty,umllinkpropertyrolelink).
representedby(anymutualproperty,umlassociationpropertyroleassociation).
representedby(anymutualproperty,umlclasspropertyroleassociation).
precededby(anymutualproperty,abilitytoassociate).
relationtoclass(anymutualproperty,associatedthingsanymutualproperty).
ontologyproperty(anytransformationlaw).
precedes(anytransformationlaw,emergentbehaviour).
ontologyproperty-name(anytransformationlaw,anytransformationlaw).
representedby(anytransformationlaw,umlobjectpropertyroleoperation).
representedby(anytransformationlaw,umlclasspropertyroleoperation).
representedby(anytransformationlaw,umloperationpropertyroleoperation).
relationtoclass(anytransformationlaw,changeablethingsanytransformationlaw).
ontologyproperty(emergentbehaviour).
ontologyproperty-name(emergentbehaviour,emergentbehaviour).
precededby(emergentbehaviour,anytransformationlaw).
relationtoclass(emergentbehaviour,systemthingsemergentbehaviour).
namedrepresentedphenomenon(umlobjectpropertyroleoperation).
namedrepresentedphenomenon-rolename(umlobjectpropertyroleoperation,operation).
describesconstruct(umlobjectpropertyroleoperation,umlobject).
namedrepresentedphenomenon-maxcardinality(umlobjectpropertyroleoperation,-1).
represents(umlobjectpropertyroleoperation,anytransformationlaw).
namedrepresentedphenomenon-mincardinality(umlobjectpropertyroleoperation,0).
namedrepresentedphenomenon(umlaggregationpropertyrolepartwholerelation).
namedrepresentedphenomenon-rolename(umlaggregationpropertyrolepartwholerelation,partwholerelation).
describesconstruct(umlaggregationpropertyrolepartwholerelation,umlaggregation).
namedrepresentedphenomenon-maxcardinality(umlaggregationpropertyrolepartwholerelation,1).
represents(umlaggregationpropertyrolepartwholerelation,partwholerelation).
namedrepresentedphenomenon-mincardinality(umlaggregationpropertyrolepartwholerelation,1).
namedrepresentedphenomenon(umlaggregationclassroleaggregate).
namedrepresentedphenomenon-rolename(umlaggregationclassroleaggregate,aggregate).
describesconstruct(umlaggregationclassroleaggregate,umlaggregation).
namedrepresentedphenomenon-maxcardinality(umlaggregationclassroleaggregate,1).
represents(umlaggregationclassroleaggregate,compositethings).
namedrepresentedphenomenon-mincardinality(umlaggregationclassroleaggregate,1).
namedrepresentedphenomenon(umlassociationpropertyroleassociation).
namedrepresentedphenomenon-rolename(umlassociationpropertyroleassociation,association).
describesconstruct(umlassociationpropertyroleassociation,umlassociation).
namedrepresentedphenomenon-maxcardinality(umlassociationpropertyroleassociation,1).
represents(umlassociationpropertyroleassociation,anymutualproperty).
namedrepresentedphenomenon-mincardinality(umlassociationpropertyroleassociation,1).
namedrepresentedphenomenon(umlobjectclassroleclass).
namedrepresentedphenomenon-rolename(umlobjectclassroleclass,class).
describesconstruct(umlobjectclassroleclass,umlobject).
namedrepresentedphenomenon-maxcardinality(umlobjectclassroleclass,1).
represents(umlobjectclassroleclass,allthings).
namedrepresentedphenomenon-mincardinality(umlobjectclassroleclass,1).
namedrepresentedphenomenon(umlpropertyclassroleclass).
```

Appendix D

UEML 2.0 Template

UEML Template
Version 1.2

Document type:	Working document
Domain/Task/Topic:	DEM / UEML / Approaches
Version:	2
Date:	2005-09-04
Status:	1st iteration
Authors:	Andreas L Opdahl
Distribution list:	DEM
Document history:	

Intended use of the template: Fill in the entries of this template once for each modelling construct in the language you are describing. Delete the lead texts (in blue like this one) and the short explanations (in red). This version of the template is not fully formal, and it is ok to deviate from the proposed structure if you need. It is also ok to add comments, explanations and TBDs anywhere as you go along.

Additional resources: The following resources are available to make the template easier to use:

- More detailed guidelines and examples are available in the associated "UEML Template Tutorial".
- Existing construct descriptions, in particular descriptions of all the constructs in UEML version 1.0, which are particularly useful because of their simplicity. An "Introduction to the BWW-representation model and Bunge's ontology" on which the template is based, is also available.
- References [1–2] are important precursors to the template.

Change list: Changes from versions 1.0 and 1.1 of the template are listed at the end of this document, along with an indiciation of some planned future developments. There is no need to update existing construct descriptions to reflect the new version of the template.

Does the template look big? Most modelling constructs will be rather simple to define, and new construct definitions can defined in terms of existing ones using the *builds on* entries described below. A few constructs, particularly the ones that represent dynamic behaviour may become complex, but they often resemble dynamic constructs from other languages, so there is even potential for reuse across languages.

Synopsis: Brackets mean [OPTIONAL INFORMATION], parentheses with a star mean (ZERO OR MORE TIMES) *, bars mean EITHER THIS | OR THAT etc. These are suggestions only. You may fill in the entries in other ways if you prefer. A stricter documents structure/syntax will be provided when we move to XML or descendants.

["<QUOTE FROM 'OFFICIAL' LANGUAGE DEFINITION>"
<REFERENCE TO 'OFFICIAL' LANGUAGE DEFINITION>]

[< BRIEF INTRODUCTION>]

1. Preamble

The *Preamble* provides general information about the modelling construct, such as the construct name, related constructs, the diagram types and language it belongs to, as well as acronyms and external resources.

Builds on

(<NAME OF OTHER CONSTRUCT IN THIS LANGUAGE>) *

Later entries in the template can then be filled in with

As for <OTHER CONSTRUCT> **but with … replacing/modifying …**

or with

As for <OTHER_CONSTRUCT>

or marked empty with

None

Built on by

(<NAME OF OTHER CONSTRUCT IN THIS LANGUAGE>) *

Construct name

<NAME OF THIS CONSTRUCT>

Alternative construct names

(<ALTERNATIVE NAME THAT IS SOMETIMES USED>) *

Related, but distinct construct names

- (<OTHER CONSTRUCT NAME>: <EXPLANATION OF DIFFERENCE>) *

Related terms

- (<RELATED TERM>: <EXPLANATION OF TERM>) *

Language

<LANGUAGE NAME>, <VERSION NUMBER> (<ACRONYM>), <PRIMARY URI>.
[((<PRIMARY REFERENCE>.) *
(<OTHER URIS AND REFERENCES>.) *
(<ORGANISATIONS>.) *)]

Diagram type

(<NAME OF DIAGRAM TYPE IN THIS LANGUAGE>) *

2. Presentation

This section describes the *visual presentation* of the modelling construct. Presentation issues include lexical information (such as icons, line styles), syntax (how this and other modelling constructs connect to form diagrams and repositories) and pragmatics (in particular layout conventions). The *Presentation* section of the template is still informal, because we have given priority to the (presumably more difficult) *Representation* section that follows.

Builds on

(<NAME OF OTHER CONSTRUCT IN THIS LANGUAGE>) *

Built on by

(<NAME OF OTHER CONSTRUCT IN THIS LANGUAGE>) *

Icon, line style, text

For a construct that is represented as a node:

<DRAWING OF ICON>
<EXPLANATION OF HOW PRESENTATION DEPENDS ON ATTRIBUTE VALUES>

For a construct that is represented as an edge:

<DRAWING OF LINE STYLE>
<EXPLANATION OF HOW PRESENTATION DEPENDS ON ATTRIBUTE VALUES>

User-definable attributes

- (<CARDINALITY CONSTRAINT> <ATTRIBUTE NAME>
  [: <ATTRIBUTE TYPE> [= <DEFAULT VALUE>]]
  (<OPTIONAL DESCRIPTION>)) *

Where < CARDINALITY CONSTRAINT> is:

(<MINCARD> : <MAXCARD>)

Relations to other constructs

- <FULL CARDINALITY CONSTRAINT> <RELATION NAME>
  : <OTHER CONSTRUCT NAME>
  (<OPTIONAL DESCRIPTION>)

Where <FULL CARDINALITY CONSTRAINT> is:

(<MINCARD> : <MAXCARD> [**rev** <MINCARD> : <MAXCARD>])

Diagram layout conventions

- (<DESCRIPTION OF LAYOUT CONVENTION>) *

Other usage conventions

- (<DESCRIPTION OF OTHER USAGE CONVENTION>) *

3. Representation

This section describes that instantiation level, classes, properties and kinds of dynamic behaviour that this modelling construct can be used to represent.

Builds on

(<NAME OF ANOTHER CONSTRUCT IN THIS LANGUAGE>) *

Built on by

(<NAME OF ANOTHER CONSTRUCT IN THIS LANGUAGE>) *

Instantiation level

Type level | Instance level | Both type and instance level

Classes of things

<CARDINALITY CONSTRAINT> “<CLASS ROLE NAME>”

played by <ONTOLOGY CLASS >
[(<EXPLANATION OF CLASS>)]
Also represented by: <OPTIONAL LIST OF OTHER CONSTRUCTS>

Where <ONTOLOGY CLASS> can be a single:
<ONTOLOGY CLASS NAME>
or, in general, an intersection class of the form:
<ONTOLOGY CLASS NAME> (& <ONTOLOGY CLASS NAME>) *

Properties (and relationships)

[<CARDINALITY CONSTRAINT>] "<PROP ROLE NAME>"
played by <ONTOLOGY PROPERTY NAME>
[**Type:** <PROPERTY DATA TYPE> [**, default value:** <DEFAULT VALUE>]]
(**Belongs to:** <FULL CARDINALITY CONSTRAINT> <CLASS ROLE NAME>
[**in role** <ROLE NAME>]) *
(**Sub-property of:** <FULL CARDINALITY CONSTRAINT> <PROP ROLE NAME>
[**in role "**<ROLE NAME>"]) *
[[**State** | **Transformation**] **law:** <LAW DESCRIPTION>]
[(<EXPLANATION OF PROPERTY>)]
[**Also represented by:** (<ANOTHER CONSTRUCT IN THE SAME LANGUAGE>) *]

Where <ROLE NAME> is:
<DOWNWARDS ROLE NAME> [/ <UPWARDS ROLE NAME>]

Behaviour

Existence | State | Event | Process

REPRESENTED STATE ENTRIES (ONE IF STATE, ONE OR MORE IF EVENT OR PROCESS):

"<STATE ROLE NAME>" [**played by** <OPTIONAL ONTOLOGY STATE>]
(**Defining property:** <PROPERTY NAME>) *
State constraint: <STATE CONSTRAINT DESCRIPTION>
[, **constrained by** <PROPERTY NAME>]
[Also represented by: (<ANOTHER CONSTRUCT IN THE SAME LANGUAGE>) *]

Where <PROPERTY NAME> is:
"<PROPERTY ROLE NAME>" [**of** "<CLASS ROLE NAME>"]

REPRESENTED EVENT ENTRIES (ONE IF EVENT, TWO OR MORE IF PROCESS):

"<EVENT ROLE NAME>" [**played by** <OPTIONAL ONTOLOGY EVENT>]
From state: "<STATE ROLE NAME>"
To state: "<STATE ROLE NAME>"
Trigger: <TRIGGER CONDITION DESCRIPTION>
Condition: <EVENT CONDITION DESCRIPTION>
Action: <ACTION DESCRIPTION>
[, **effected by** <PROPERTY NAME>]
[Also represented by: (<ANOTHER CONSTRUCT IN THE SAME LANGUAGE>) *]

Modality (permission, recommendation etc)

Regular assertion |

Permission of "<ACTOR CLASS ROLE>"
[, **issued by** "<INTENTIONAL CLASS ROLE>"] |
Mandate of "<ACTOR CLASS ROLE>"
[, **issued by** "<INTENTIONAL CLASS ROLE>"] |
Intention of "<INTENTIONAL CLASS ROLE>" |
Obligation of "<INTENTIONAL CLASS ROLE>" |
Belief of "<INTENTIONAL CLASS ROLE>" |
Knowledge of "<INTENTIONAL CLASS ROLE>"

4. Open Issues

- (<OPEN ISSUE>: <DESCRIPTION AND DISCUSSION>) *

Change List

Changes 1.1 → 1.2: Split the template into a pure" template and a separate "Template Tutorial". Introduced a *Preamble* entry for modelling constructs that *build on* other constructs. Other changes concern mainly the *Semantics* section, which has become more precisely specified. There is no need to update existing analyses to reflect the new template version.

Changes 1.0 → 1.1: Event definitions are more detailed now, comprising a trigger, a condition and an action sub-entry. Otherwise, this revision is mainly to fix minor inaccuracies and provide better explanations of some entries in the *Semantics* section.

References

[1] Andreas L. Opdahl and Brian Henderson-Sellers. A Template for Defining Enterprise Modelling Constructs. *Journal of Database Management (JDM)* 15(1). Idea Group Publishing, 2004.

[2] Andreas L. Opdahl and Brian Henderson-Sellers. Template-Based Definition of Information Systems and Enterprise Modelling Constructs. Chapter 6 in *Ontologies and Business System Analysis*, Peter Green and Michael Rosemann (eds.). Idea Group Publishing, 2005.